DEEP CNN AND MLP-BASED VISION SYSTEMS FOR ALGAE DETECTION IN AUTOMATIC INSPECTION OF UNDERWATER PIPELINES

Edgar Eduardo Medina Castañeda

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientadores: Mariane Rembold Petraglia
José Gabriel Rodriguez Carneiro Gomes

Rio de Janeiro
Outubro de 2017

DEEP CNN AND MLP-BASED VISION SYSTEMS FOR ALGAE DETECTION IN AUTOMATIC INSPECTION OF UNDERWATER PIPELINES

Edgar Eduardo Medina Castañeda

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

Prof. Mariane Rembold Petraglia, Ph.D.

Prof. José Gabriel Rodriguez Carneiro Gomes, Ph.D.

Prof. Felipe Maia Galvão França, Ph.D.

Dr. André de Almeida Maximo, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 2017

Medina Castañeda, Edgar Eduardo

Deep CNN and MLP-based Vision Systems for Algae Detection in Automatic Inspection of Underwater Pipelines/Edgar Eduardo Medina Castañeda. – Rio de Janeiro: UFRJ/COPPE, 2017.

XIV, 74 p.: il.; 29, 7cm.

Orientadores: Mariane Rembold Petraglia

José Gabriel Rodriguez Carneiro Gomes

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2017.

Referências Bibliográficas: p. 65 – 72.

1. Convolutional Neural Networks. 2. Algae Detection System. 3. Computer Vision. I. Petraglia, Mariane Rembold *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

“The mind that opens to a new idea never returns to its original size.”
(Albert Einstein)

To my parents and my sister.

Acknowledgements

This dissertation would not have been possible without the support of my family, for inspiring me and encouraging me to follow my dreams. I am especially grateful to my dear parents, who supported me financially.

I would like to express my special appreciation and thanks to my advisors Ph.D. Mariane Rembold Petraglia and Ph.D. José Gabriel Rodriguez Carneiro Gomes, for guiding me and allowing me to grow in my skills, also have been a tremendous mentors for me and great persons to work with.

Thanks to my dear friends and roommates, for listening, offering me advice, and supporting me throughout this entire process. Special thanks to my friends Roberto Campos, Francinei Gomes, Roberto Moura and Felipe Petraglia from Analog and Digital Signal Processing Laboratory (PADS) for sharing their knowledge and debating about better solutions. Special acknowledgements also to my dear friends Natanael Moura, Nako Sung, Christian Alvarado, Juvissan Aguedo, Victor Rodrigues, Andres Bedoya and Eduardo Osorio.

Thanks to Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for financial support; also thanks to Electrical Engineering Program (PEE) and to Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia (COPPE) to help me in paperwork processes.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

SISTEMAS DE VISÃO BASEADOS EM CNN PROFUNDA E MLP PARA DETECÇÃO DE ALGAS NA INSPEÇÃO AUTOMÁTICA DE DUTOS SUBAQUÁTICOS

Edgar Eduardo Medina Castañeda

Outubro/2017

Orientadores: Mariane Rembold Petraglia
José Gabriel Rodriguez Carneiro Gomes

Programa: Engenharia Elétrica

Redes neurais artificiais, como o perceptron multicamada (MLP), têm sido cada vez mais empregadas em várias aplicações. Recentemente, as redes neurais profundas (deep neural networks), especialmente as redes neurais convolutivas (CNN), receberam atenção considerável devido à sua capacidade de extrair e representar abstrações de alto nível em conjuntos de dados. Esta dissertação descreve um sistema de inspeção automático baseado em algoritmos de aprendizado profundo (deep learning) e visão computacional para detecção de algas em dutos submarinos. O algoritmo proposto compreende uma rede CNN ou MLP, seguida de uma fase de pós-processamento que opera em domínios espaciais e temporais, empregando agrupamento de posições de detecção vizinhas e um buffer das regiões de interseção ao longo dos quadros. Os desempenhos de MLP, empregando diferentes descritores, e os classificadores CNN são comparados em cenários do mundo real. Mostra-se que a fase de pós-processamento diminui consideravelmente o número de falsos positivos, resultando em uma taxa de acerto de 99,39%.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DEEP CNN AND MLP-BASED VISION SYSTEMS FOR ALGAE DETECTION IN AUTOMATIC INSPECTION OF UNDERWATER PIPELINES

Edgar Eduardo Medina Castañeda

October/2017

Advisors: Mariane Rembold Petraglia
José Gabriel Rodriguez Carneiro Gomes

Department: Electrical Engineering

Artificial neural networks, such as the multilayer perceptron (MLP), have been increasingly employed in various applications. Recently, deep neural networks, specially convolutional neural networks (CNN), have received considerable attention due to their ability to extract and represent high-level abstractions in data sets. This work describes a vision inspection system based on deep learning and computer vision algorithms for detection of algae in underwater pipelines. The proposed algorithm comprises a CNN or a MLP network, followed by a post-processing stage operating in spatial and temporal domains, employing clustering of neighboring detection positions and a region interception framebuffer. The performances of MLP, employing different descriptors, and CNN classifiers are compared in real-world scenarios. It is shown that the post-processing stage considerably decreases the number of false positives, resulting in an accuracy rate of 99.39%.

Contents

List of Figures xi

List of Tables xiv

1 Introduction 1
- 1.1 Problem Context 1
- 1.2 Project Motivation 2
- 1.3 Objectives 2
- 1.4 Contributions 3
- 1.5 Dissertation Outline 3
- 1.6 Related Works 3

2 Image Processing and Feature Extraction 5
- 2.1 Image Processing 5
- 2.2 Features extraction 6
  - 2.2.1 Color-based features 6
  - 2.2.2 Entropy-based features 6
  - 2.2.3 Wavelets-based features 6
  - 2.2.4 Local Binary Patterns 7
  - 2.2.5 Gray-Level Co-occurrence Matrix 9
  - 2.2.6 Hu-moments-based 9

3 Artificial Neural Networks 10
- 3.1 Machine Learning Basics 10
  - 3.1.1 Types of Machine Learning Algorithms 10
  - 3.1.2 Generalization Issues 12
  - 3.1.3 Logistic Regression 12
  - 3.1.4 Softmax Regression 14
- 3.2 Gradient descent optimization algorithms 15
- 3.3 Multi-Layer Perceptron 17
  - 3.3.1 Perceptron 18
  - 3.3.2 Multilayer Perceptron 18

3.4 Convolutional Neural Networks 20
3.4.1 Convolution Layers 21
3.4.2 Pooling Layers 22
3.4.3 Convolutional Neural Networks 24
3.5 Regularization 25
3.6 Backpropagation Algorithm 28
3.7 Initialization 31

4 Methodology 32
4.1 Features Extractor Algorithms 32
4.2 Neural Networks System 34
4.2.1 MLP-based System 35
4.2.2 CNN-based System 37
4.3 False Positives Reduction 38
4.3.1 Spatial Algorithm 38
4.3.2 Temporal Algorithm 39

5 Results and Discussions 43
5.1 Design Analysis 43
5.1.1 Loss Function 44
5.1.2 Activation Function 44
5.1.3 Regularization 45
5.1.4 Optimizer 46
5.1.5 Initialization 46
5.1.6 CNN hyperparameters 47
5.2 Performance Analysis 49
5.2.1 MLP Topologies 50
5.2.2 CNN Topologies 52
5.2.3 CNN+MLP Topologies 54
5.3 Other Metrics 56
5.3.1 Relevant inputs 56
5.3.2 Score metrics 58
5.4 Systems Comparison 59
5.4.1 False Positives Reduction 59
5.4.2 MLP-based System 60
5.4.3 CNN-based System 62

6 Conclusions and Future Work 63

Bibliography 65

A Algae Database **73**

B Looking at layers **74**

List of Figures

2.1 Wavelet decomposition block. In (a), Basic wavelets filters bank block for two-level-decomposition of the input, and in (b), the nested blocks to obtain the four-level decomposition from image using the subband S_2. 7
2.2 3x3 pixels window as kernel extracted from a grayscale image (a), to obtain LBP output image (b). 8

3.1 examples of regression (top) and classification (bottom) problems, including three solution types that are commonly found: underfit (left), good (center) and overfit (right) solutions. 11
3.2 Training and validation error curves indicating situations of underfitting, overfitting, and adequate generalization. 13
3.3 Artificial neuron. 18
3.4 General architecture of a multi-layer neural network. 20
3.5 Neural network solution for a classification problem. 20
3.6 Connections between $H_1 \times W_1 \times D_1$ input volume and $H_2 \times W_2 \times D_2$ output volume. 23
3.7 Connections between input and output volumes, illustrating downsampling by a factor of two at the pooling layer (i.e. at the convolutional layer output). 24
3.8 CNN topology using two stages (convolutional layer and pooling layer) repeated twice, thus composing two CNN layers, immediately followed by an additional pooling layer, flattening, and three fully-connected layers. 25

4.1 MLP-based system. 36
4.2 CNN-based System. 38

4.3 Illustration of spatial analysis for false positive removal. In (a), the image obtained from classifier with 30 positive algae detection windows (indicated as P); step space is given by two cells. In (b), First step of the algorithm creates labels for positive algae detection windows based on neighborhood detections. In (c), Second step of the algorithm defines regions with algae and eliminates false positives. . . 39
4.4 Temporal analysis for removing false positives in current frame based on detected regions of previous frames. In (a), Image from current frame is analyzed employing 3 consecutive frames, (b) detected regions after spatial analysis and (c) temporal analysis using 3 time steps in the framebuffer. In (d), Temporal analysis after removing the false positive regions. 42

5.1 Activation function comparison: training (dashed lines) and validation (solid lines) classification error sequences using sigmoidal (green), hyperbolic tangent (red), or ReLU (blue) activation functions. 45
5.2 Effects of regularization at the convolutional layers of a neural network with CNN+MLP topology: training (dashed lines) and validation (solid lines) classification error sequences using no regularization (cyan); L2 and GAP (magenta); and L2, GAP and batch normalization (black). 46
5.3 Comparison of momentum (green), RMSprop (blue), and Adam (red) training results for a CNN+MLP topology. Training and validation classification error are presented in dashed and solid lines respectively. 47
5.4 Comparison between Xavier and uniform initialization methods. In (a), Xavier and Uniform initialization for a MLP topology. In (b), Xavier and Uniform initialization for a CNN topology. Training and validation classification error are presented in dashed and solid lines respectively. 48
5.5 Kernel size variation at the first (a) and second (b) convolutional layers of a two-layer CNN. In (a), the average median error is 2.48%. In (b), the average median error is 2.08%. 48
5.6 Different error results obtained by varying the number of filters at the first or second convolutional layer of a two-layer CNN. In (a) and (c), the number of filters is held constant at the second layer (16 and 32, respectively). The average median error values are 2.83% and 2.91%, respectively. In (b) and (d), the number of filters at the first layer is constant (16 and 32, respectively). The average median error values are 3.23% and 1.95%, respectively. 50

5.7 Different error results obtained by varying neural network depth, for CNN topologies (a) and for CNN+MLP topologies (b). For CNNs (a), the average median error is 1.93%, and for CNN+MLP topologies the average median error is 1.11%. 51
5.8 Relevant inputs for trained models MLP1, MLP3 and MLP6, obtaining a model accuracy of 93.04%, 87.67% and 95.71% respectively. . . 57
5.9 Relevant features after global pooling layer for topology CNN+MLP 18, model accuracy 99.47%. 58
5.10 Performance and runtime of the algae detection system for different values of the window shift step. In (a), Region error percentage. In (b), Region error increase using 1-pixel shift as reference. In (c), False positive rate. In (d), Algorithm runtime. 60
5.11 Performance and runtime of the algae detection system for different frame buffer sizes. False positive rate and algorithm runtime for (a) and (b) respectively. 61
5.12 Examples of algae detection obtained MLP-based system with 8-pixels spatial shift, minimum cluster size equal to 8 and 2 images in frame buffer. 61
5.13 Examples of algae detection obtained CNN-based system employing 8-pixels spatial shift, minimum cluster size equal to 8 and 2 frames into frame buffer. 62

A.1 Negative (a) and positive (b) samples from image database without data augmentation. 73

B.1 Looking at the layers using a positive input sample (a). The first (b), second (c) and third (d) convolutional layers are shown. 74

List of Tables

3.1 Activation functions used in the present work. 19

5.1 Accuracy comparison among MLP topologies 1 to 6. 52
5.2 Accuracy and runtime comparisons among 23 designed CNNs. Standard deviation values are provided next to mean error values (five folds). 53
5.3 Accuracy and runtime comparisons among 20 designed CNN+MLP topologies. Standard deviation values are provided next to mean error values (five folds). 55
5.4 Comparison of false positive rate for selected models. 58

Chapter 1

Introduction

This chapter explains the motivation for this project, and it introduces the problem of classifier development for underwater pipeline inspection. The objectives, contributions, and structure of this dissertation are outlined.

1.1 Problem Context

Underwater pipeline inspection has become increasingly challenging with the expansion of underwater field exploration [1], [2]. Automatic inspections are often performed by Remotely Operated Vehicles (ROVs) and Autonomous Underwater Vehicles (AUVs), which carry sensors and cameras and are handled either through cable connections from the vehicle to the operators or through radio control [2], [3]. AUVs decrease the interaction between human and inspection procedure, which is due to the generalization capacity that is expected from the type of system that is proposed in this work.

An accurate and efficient inspection system can prevent leaks and environmental problems. Often underwater pipelines accumulate sand and algae on their surfaces, which can hide damages. Therefore, it is important that the inspection system recognizes and notifies the presence of algae and sand [4], [5]. In particular, in vision-based systems, algae present a large diversity of shapes, colors and textures [6], which vary with different external conditions such as their constant movements due to water flow caused by sea current and turbulence generated by ROVs [1].

Classic neural network techniques, such as the multilayer perceptron (MLP), are strongly dependent on feature extraction methods. Recently, deep learning algorithms have been developed to iteratively extract their own features from original data. A recent deep learning technique, namely deep convolutional neural network (CNN) [7], was applied in this work. The neural network architectures and parameters that result in optimal classifier performance are selected. MLP algorithms employing different features, such as color information, wavelet coefficients statis-

tics, local binary patterns (LBP), Hu moments, entropy and gray-level co-occurrence matrix (GLCM), are also described. The results obtained using MLP and deep CNN architectures are compared under various real-world scenarios.

This work presents artificial neural network (ANN) based algorithms developed for the automatic detection of the presence of algae in underwater pipelines.

1.2 Project Motivation

In complex computer vision classification problems whose solutions traditionally depend on feature extraction methods, which is the case of algae detection based on texture analysis, deep learning techniques [8] have had a major impact over the last ten years [9]. In conventional feature extraction methods, design choices for feature extraction are made by experts. For those tasks, the experts usually rely on previous studies, or on topic-specific knowledge, or both. Deep learning methods, on the other hand, allow for automatic feature extraction. The classifying systems thus developed do not require expert-defined features and, in addition to that, consistently outperform conventional classifiers with respect to test accuracy, test mean-squared error, and so forth.

1.3 Objectives

The main objective of this research is to develop a system that is able to detect the presence of algae on the surface of subsea pipelines employing machine learning and image processing techniques. More specifically, the sub-objectives that emerge from main objective are:

- Generating a manually annotated image database from the underwater pipeline inspection videos that are available;
- Designing feature extractors that lead to a suitable balance between processing time and test classification accuracy;
- Making the classification system robust to different external conditions in which pipelines are commonly found;
- Comparing multi-layer perceptron networks and convolutional neural networks with respect to test classification accuracy;
- Designing post-processing algorithms to reduce false-positive classification ratios.

1.4 Contributions

In this work, we propose methods for training neural network classifiers for the algae detection problem. Several different neural network models are designed using the proposed methods, and these models are compared both objectively (with respect to test classification accuracy) and subjectively (with respect to visually assessed performance in non-annotated video segments). For multi-layer perceptron classifiers, several feature extractors are designed, aiming at feature extraction methods that lead to a low false-positive ratio. For both the multi-layer perceptron and the convolutional neural network classifiers, a region-based post-processing algorithm is designed. The results indicate that neural networks (multi-layer perceptron and convolutional) achieve reasonable performance in the automatic inspection problem of algae detection, even if the image is highly noisy and blurred. For neural network training, we created a proprietary manually annotated algae image database, using proprietary pipeline inspection videos. Additional information about the algae image database is provided in Appendix A.

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows. We present theoretical background of image processing for feature extraction in Chapter 2, and neural network theoretical background in Chapter 3. In Chapter 4, we present conventional feature extraction, as well as a post-processing algorithm. The post-processing algorithm is based on spatial and temporal analyses, and it aims at improving false-positive classification results. In Chapter 5, we present experimental results and a comparison between MLP and CNN classification performances. Concluding remarks and a brief discussion about future work topics are given in Chapter 6. Appendix A lists samples from the Algae dataset, which is used in the experimental performance assessment. Appendix B presents details about the libraries that are used in the present work.

1.6 Related Works

Related algorithms regarding pattern recognition and image classification include texture description, feature extraction, and machine learning techniques. In [10], an LBP histogram selection approach for color texture classification was presented. In [11], a rotation-invariant texture extraction technique using principal component analysis (PCA) and dual-tree complex wavelet transform (DT-CWT) was proposed. In [12], the authors suggested classification features based on the multi-scale wavelet

transform of the original image, or features based on a smooth cubic spline surface computed from the original image. In [13], a comparison between curvelet and wavelet texture features was presented. In [14], the authors carried out a comparative study of texture detection and texture classification algorithms using Gabor filters, Laws masks, ring/wedge filters, GLCM, and autoregressive image models. In [15], the binary rotation invariant and noise-tolerant euclidean (BRINE) metric feature was presented for multi-view face recognition. In [16], a method for plant species identification is reported. The method is based on common flower features such as color, texture, and shape, in addition to fractal dimension information.

In [17] and [18], the authors described other applications for machine learning, such as bovine tuberculosis detection and pedestrian detection. Marchi and others [19] present a deep recurrent neural network based on autoencoders applied to acoustic novelty detection. Bergado [20] presents a master's thesis about deep learning applied to urban scene classification. In that dissertation, he tried different conventional and deep neural network topologies in order to develop a recurrent convolutional neural network. Hafemann [21] presents a master's thesis about texture classification using deep CNNs.

Regarding underwater targets, Qin and others [22] used convolutional layers and a linear SVM classifier for fish recognition. Villon and others [23] presented results for coral reef fish detection and recognition in video sequences. They compared conventional machine learning methods with deep learning. Cao and others [3] proposed a feature extraction system, based on deep-learning techniques, using a stacked autoencoder. Like [16], Lee and others [24] also work in the context of plant classification. They used CNNs to learn unsupervised feature representations for 44 plant species.

Many works about CNNs (regularization, initialization, architecture selection, and so on) were published in the last decade [9]. Srivastava and others [25] proposed *dropout*, which is a novel regularization technique. Dropout is widely used in deep learning nowadays, and it is easily applicable to deep CNN training. Ioffe and others [26] introduced *batch normalization*, which is another important deep learning tool, as it significantly mitigates internal covariate shift problems in the mini-batch training mode. To save computational resources in deep neural network training, new gradient-based optimization techniques have been published [27], [28], [29] and are widely used nowadays.

To support video object tracking tasks, algorithms using different methods have been proposed [30], [31], [32]: an image pre-processing and centroid-based method, a method combining *camshift* and Kalman prediction, and a method combining color analysis and Hu moments. Additional image pre-processing work related to underwater image analysis was proposed by Yang and others [33].

Chapter 2

Image Processing and Feature Extraction

This chapter describes image processing techniques aimed to image enhancement and feature extraction used in the machine learning classification task.

2.1 Image Processing

In this work image processing was employed to enhance image using several techniques that provide more detailed information from images such as edge enhancement, spatial filtering, background removal and color-based histogram equalization. The purposes of employing these techniques are described below:

- *Edge enhancement*: In underwater inspection tasks commonly the captured images are blurred and the shapes of the objects are missing and/or are confused with the background due to the non-controllable environment. These set of techniques are applied to recover the edges in the image and can be used as a features extraction stage [34], [35];

- *Spatial filtering*: in order to help in the detection of objects, edge enhancement does not always provide relevant information to generate features because background artifacts are enhanced as well. Spatial filtering removes high gradients (intensity variations) throughout the image [35], [34];

- *Color-based histogram equalization*: Depending on the scenario, RGB color domain is usually not the best option for image processing [35]. HSV color domain is normally used to separate **hue** and **saturation** channels from **lightness**, and histogram equalization enhances color information in these channels, distributing the color throughout the spectrum in order to reduce the influence of **lightness** over images [35], [34], [36];

- *Background removal*: Background usually becomes a problem when feature extractor is based on texture or edges components [34], [36]. Simple segmentation techniques, such as K-means, can be employed to remove these artifacts, because only two clusters, the background and foreground image sections, are necessary [35], [34], [36].

2.2 Features extraction

The feature extractor, also called descriptor, provides non-redundant information with a smaller dimensionality size than the original input. When input data is too large to be processed, the data representation can always be represented in smaller dimension data, that is, it can be transformed into a reduced set called feature vector. The selected features contain relevant information from original data and this reduced representation can be used, instead of the complete initial data, to perform certain machine learning tasks. In this work, the descriptor is mainly based on extraction of texture, color and shape.

2.2.1 Color-based features

Color-based features are useful when input images contain high color variations or the pixel values distribution is a wide Gaussian function (color histogram with a wide color spectrum). This information needs to be interpreted as numbers and usually statistical values are collected from histograms or even the histogram is used as the feature vector [34].

2.2.2 Entropy-based features

Color or another spatial information are not directly manageable to compute the feature vector. This is why statistical parameters are computed and commonly used instead of the raw information. Mean, variance, skew and others are first-order statistical parameters, entropy also measures the unpredictability of the state, disorder or information in a determined group data. These parameters are employed to collect information from color or spatial features of the image [34], [36].

2.2.3 Wavelets-based features

Wavelets are powerful and very useful techniques in image and signal processing. They can be applied as texture extractor due to the fact that the wavelet property of separating high and low frequencies contributes to the segmentation of the high and low intensity variations in the pixels. In this work, Daubechies 2 wavelet **(db2)** was

employed in the experiments due to its asymmetric and orthogonal properties [34], [36]. Figure 2.1a depicts the complete filter bank decomposition employed for the extraction of texture features, where **I(x,y)** is the input image to be decomposed by wavelet filters H_1 and H_2. The outputs $S_{n,1}$, $S_{n,2}$, $S_{n,3}$ and $S_{n,4}$ represent the second level decomposition of the input. If the input is the original image, the outputs will be the second level decomposition of the image.

If the input is $S_{n,2}$ the outputs will correspond to a subset of the second level decomposition of $S_{n,2}$ or the fourth level decomposition of the original image (using only the $S_{n,2}$). This nested representation is shown in Figure 2.1b.

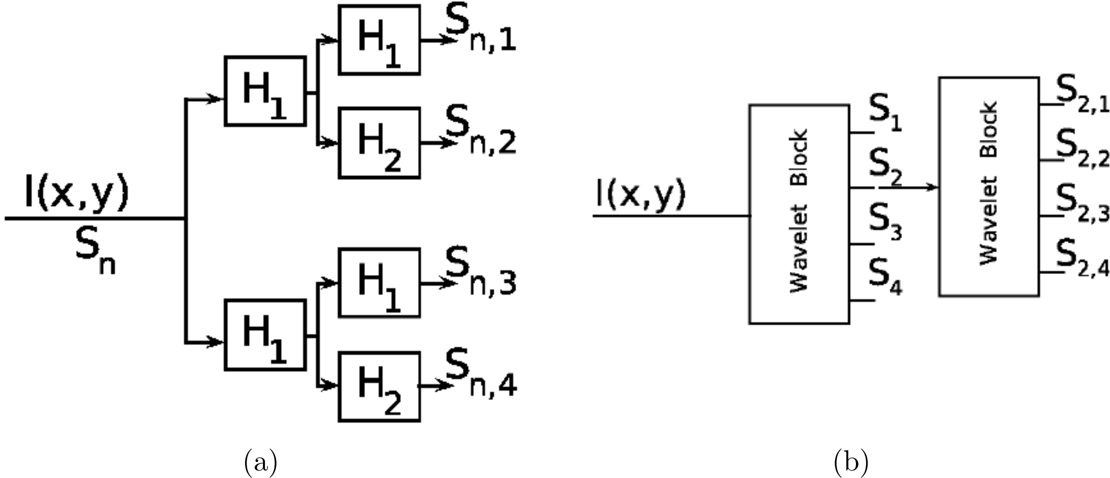


Figure 2.1: Wavelet decomposition block. In (a), Basic wavelets filters bank block for two-level-decomposition of the input, and in (b), the nested blocks to obtain the four-level decomposition from image using the subband S_2.

Useful information for this project was observed up to the third decomposition level, black output images are obtained in upper levels for some subbands. A correlation analysis was then employed to determined the most relevant subbands with the best image representations [34], [36]. This analysis is detailed in Chapter 5, which contains the simulation results.

2.2.4 Local Binary Patterns

Local binary patterns (LBP) was introduced in [37] and is widely used for texture extraction due to its capability to describe compactly and efficiently the texture information of the image. LBP is a translation, rotation and scaling invariant descriptor. However, each one of these invariances adds extra computational cost to the extractor, since different parameters, such as the spatial resolution operator, the quantization of angular space and the method to determine the pattern, must be set to obtain more robust features [34].

LBP is obtained by comparing the center pixel of a window with its neighbors and this pixel is replaced by a new computed number based on weighted binary code. The basic LBP is shown in Eq. (2.1) where **s**(x) is a thresholding function configured by the central point in the kernel expressed in Eq. (2.2). For example, given a 3x3 kernel from a gray image, depicted in Figure 2.2a, its thresholded value after applying Eq. (2.2) will be 10100001_2 and its weighted value for this binary code are 1+0+0+0+0+32+0+128 = 161 calculated through Eq. (2.1). This result will replace the center point in the current kernel, producing the matrix depicted in Figure 2.2b, Local Binary Pattern is computed throughout the image to obtain a LBP map. Finally, a histogram is obtained from the map LBP of the entire image, which describes the texture [37], [34].

$$LBP = \sum_{x \in [1,8]} \mathbf{s}(x) 2^{x-1} \tag{2.1}$$

$$\mathbf{s}(x) = \begin{cases} 1, & \text{if } P_x > P \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

127	200	20
35	100	107
0	95	100

(a)

127	200	20
35	**161**	107
0	95	100

(b)

Figure 2.2: 3x3 pixels window as kernel extracted from a grayscale image (a), to obtain LBP output image (b).

The basic LBP code is complemented with two local measures, the **contrast** and the **variance**, which can be obtained by different ways in order to compute the pixel intensities, such as encoding by differences from 1s and 0s or computing the four neighbor pixels in vertical and horizontal positions, aiming to reflect features correlations as well as contrast [34].

In this work scale invariant is also considered, replacing x by $x(i)$ in Eq. (2.1) where $x(i)$ is a function applied in order to consider points at greater distances from the center point. This idea is expressed in Eq. (2.3) and the basic LBP is replaced by **LBP_S** showed in Eq. (2.4), where P and R are the number of points in the circularly symmetric neighborhood and the radius of the circle, respectively [34], [36].

$$x(i) = \begin{cases} x_0 + Rcos(\frac{2\pi i}{P}) \\ y_0 + Rsin(\frac{2\pi i}{P}) \end{cases} \tag{2.3}$$

$$LBP_S(P,R) = \sum_{i \in [1,P]} \mathbf{s}(x(i))2^{i-1} \tag{2.4}$$

2.2.5 Gray-Level Co-occurrence Matrix

The co-occurrence matrix is defined over an image and contains the distribution of pixel values at a given offset. This statistical method is employed to analyze texture considering spatial relationships of pixels. If the image being analyzed contains only gray scale values, it is called Gray-Level Co-occurrence Matrix (GLCM). The GLCM function characterizes the texture of an image by comprising information of how often pairs of pixels occur in an image considering their specific values and spatial relationship. However, this matrix commonly is not used as an entirely, instead statistical measures are extracted from it to represent the information of pixels relationship [38]; for this reason GLCM is considered a second-order statistical features [34], [36].

2.2.6 Hu-moments-based

In mathematics, a moment is a specific quantitative measure of the shape of a set of points, and an image moment is a particular weighted average of the pixels. **Moments** are useful to describe information from objects. For instance, the area, centroid and orientation of a pixels group are moments and describe a layout that contains global description of a shape with invariance properties in compact representation without noise effects [34], [39].

Hu moments are translation, scale and rotation invariant feature and are composed of only the seven parameters based on previous calculated moments such as moment inertia around the image centroid and the invariant skew. Due to its robustness to represent different shapes, Hu moments are widely used in image processing and computer vision [40], [38], [41].

Chapter 3

Artificial Neural Networks

This chapter deals with neural network theoretical background that is required for a proper understanding of the methodology and results presented in this work. After a brief introduction on machine learning basics, we address the following topics: gradient optimization algorithms, multi-layer perceptrons (MLPs), deep convolutional neural networks (CNNs), regularization, application of the basic backpropagation algorithm to MLPs and CNNs, and, finally, initialization heuristics.

3.1 Machine Learning Basics

This section addresses fundamental machine learning concepts: algorithm types, generalization issues, logistic regression, and softmax regression.

3.1.1 Types of Machine Learning Algorithms

Solving machine learning problems corresponds to mapping input events into output decisions. The input events are also called patterns, and the respective correct or desirable decisions are called targets. According to the availability of targets, machine learning algorithms may be organized as follows [42], [43], [44]:

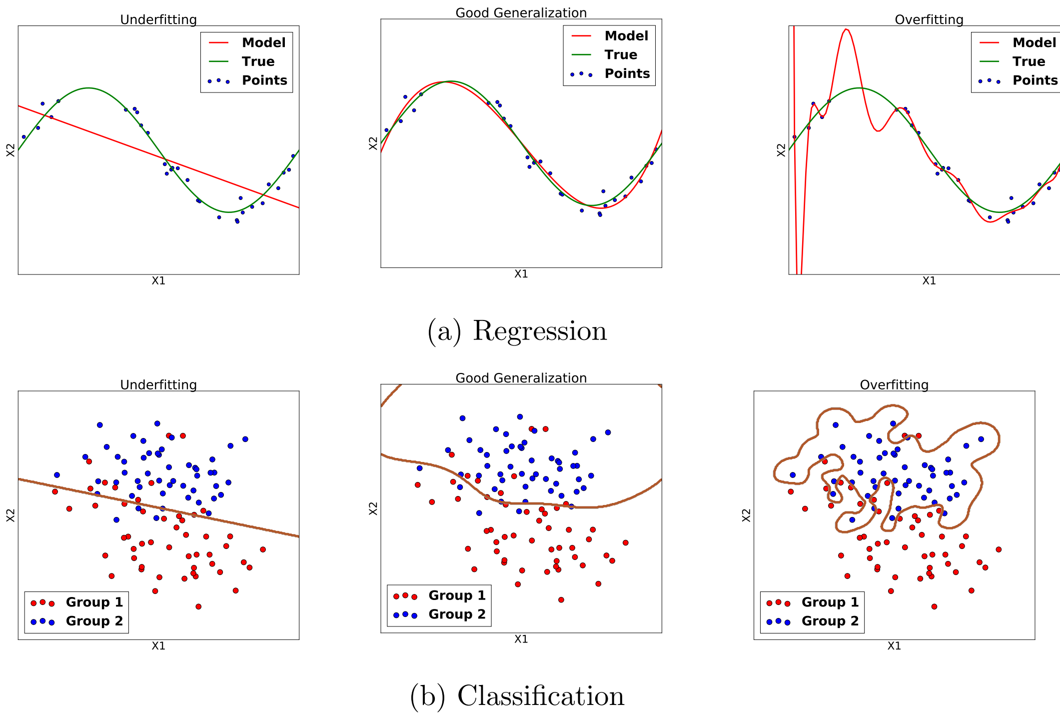
- *Supervised learning*: when a given input is referenced to a known target, the training process is called supervised learning. Every input vector (event) is mapped into an output vector, which may either represent one of several classes, or represent a continuous approximation to a real function. The output vectors are compared to target vectors during the training process, to generate error signals for parameter optimization. After training is finished, new inputs and new targets may be used for test performance assessment;

- *Unsupervised learning*: the inputs are not labeled (i.e. targets are not available), and the machine learning task is to label the data. The training proce-

dure is often referred to as clustering;

- *Reinforcement learning*: in reinforcement learning, algorithms learn how to act (generate outputs) in response to different situations (input events), as in supervised learning. In reinforcement learning, however, the impact of the actions on an external environment is assessed, which generates feedback signals that are also used in a closed-loop training process.

Machine learning tasks seek functions that approximate real vector distributions (input vectors alone, or input and target vector joint distributions). If targets are available and they correspond to two or more classes, then the machine learning task corresponds to solving a classification problem. If the targets are scalar values or vectors drawn from a continuous probability density function, then the machine learning task corresponds to solving a regression problem [42], [43], [44], [45]. Figure 3.1 illustrates regression (top) and classification (bottom) problems, including three solution types that are commonly found: underfit (left), good (center) and overfit (right) solutions. Generalization issues are discussed next.



(a) Regression

(b) Classification

Figure 3.1: examples of regression (top) and classification (bottom) problems, including three solution types that are commonly found: underfit (left), good (center) and overfit (right) solutions.

3.1.2 Generalization Issues

A machine learning system capacity relates, roughly, to the ability of achieving good performance in the representation of data not previously seen [43]. The generalization error is typically assessed through the application of the model to a test set containing data samples that were not used for training. Although training and test samples ideally correspond to information extracted from the same problem domain, and are therefore identically distributed, the test samples may generate incorrect outputs, depending on the learning model. The learning model estimates the data distribution from which the training and test samples were drawn. If the machine learning algorithm optimizes model parameters to reduce training error, then a similar error is expected for the test samples. To know whether a machine learning algorithm is performing well, one must pay attention to the training error, which must be small, and to the difference between training and test (i.e. validation) error, which must be small as well [8], [17]. Large training error or large difference between training and validation errors correspond to undesirable situations, which are described next:

- *Underfitting* : if the training error is large, then the model is not able to fit the data correctly. A large training error usually occurs if the model is too simple or if training is interrupted too early;

- *Overfitting*: a large difference between training and validation errors occurs if the model is too complex or if the number of training iterations is excessive.

To avoid underfitting and overfitting related to insufficient or excessive training, strategies to stop training at the right time have been developed. Early stopping is useful for avoiding overfitting, and it also saves computer processing time. Early stopping is usually based on the difference between training and validation errors, or on the training error standard deviation, or on additional statistical properties of the training and validation error curves [46], [47], [48], [8].

3.1.3 Logistic Regression

In supervised learning for regression, the objective function under optimization measures the average distance between outputs caused by input vectors $\mathbf{x}$ and the corresponding targets $\mathbf{y}$. The function $\mathbf{h}(\mathbf{x})$ predicts the target vector $\mathbf{y}$ based on input vector $\mathbf{x}$. In linear regression, $\mathbf{h}(\mathbf{x})$ can be represented as in Eq. (3.1), where θ is a parameter vector to be optimized, either manually or automatically. The parametrized function $\mathbf{h}_\theta(\mathbf{x})$ is also referred to as the model *hypothesis*. Machine learning algorithms automatically find a θ vector that is optimal, in the sense of

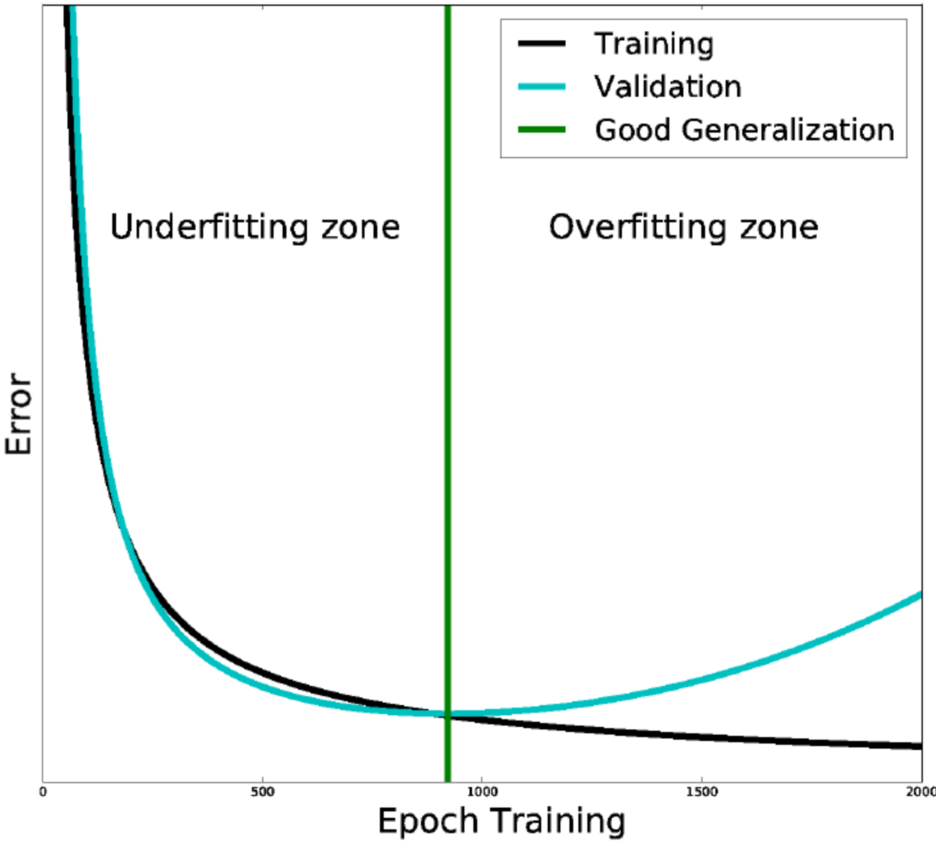


Figure 3.2: Training and validation error curves indicating situations of underfitting, overfitting, and adequate generalization.

making the model hypothesis as close as possible to the target $\mathbf{y}$ for every input vector $\mathbf{x}$. The objective function is also referred to as *loss function* or *cost function*. A cost function example based on mean squared error (MSE) is shown in Eq. (3.2) [48], [8]. In the present work, we will focus on loss function minimization algorithms based on gradient descent.

$$\mathbf{h}_\theta(\mathbf{x}) = \sum_j \theta_j x_j = \theta^T \mathbf{x} \tag{3.1}$$

$$J(\theta) = \frac{1}{2}\sum_j (h_\theta(x^j) - y^j)^2 = \frac{1}{2}\sum_j (\theta^T x^j - y^j)^2 \tag{3.2}$$

Logistic regression, in contrast to linear regression, is often applied to discrete target prediction. The logistic regression hypothesis can be thought of as the posterior probability of a vector class, given an input vector. The expressions for the probabilities of class "0" and class "1" in the binary classification case, which add up to one, are shown in Eqs. (3.3), (3.4). The hypothesis $\mathbf{h}_\theta(\mathbf{x})$, in Eq. (3.3),

corresponds to a sigmoidal function that is also known as *logistic* function. The logistic regression loss function, which is shown in Eq. (3.5), corresponds to the cross-entropy between $\mathbf{h}_\theta(\mathbf{x})$ and $\mathbf{y}$.

$$P(y=1|\mathbf{x}) = \mathbf{h}_\theta(\mathbf{x}) = \frac{1}{1+\exp\left(\theta^T\mathbf{x}\right)} \tag{3.3}$$

$$P(y=0|\mathbf{x}) = 1 - \mathbf{h}_\theta(\mathbf{x}) \tag{3.4}$$

$$J(\theta) = -\sum_j \mathbf{y}^j log(\mathbf{h}_\theta(\mathbf{x}^j)) + (1-\mathbf{y}^j) log(1-\mathbf{h}_\theta(\mathbf{x}^j) \tag{3.5}$$

To minimize $J(\theta)$, the gradient $\nabla_\theta J(\theta)$ must be computed, which is shown in scalar form in Eq. (3.6).

$$\frac{\partial J(\theta)}{\partial \theta_i} = \sum_j x_i^j (h_\theta(x^j) - y^j) \tag{3.6}$$

3.1.4 Softmax Regression

Softmax regression may be thought of as the multi-class form of logistic regression (i.e. classification with multiple-class targets). The hypothesis, which is shown in Eq. (3.7), corresponds to the posterior probability $P(\mathbf{y}=\mathbf{n}|\mathbf{x})$, for $n=1,2,\ldots,N$, where N is the number of classes taken into account in the softmax regression. As in the two-class logistic regression, the softmax regression loss function $J(\theta)$ corresponds to the cross-entropy between $\mathbf{h}_\theta(\mathbf{x})$ and $\mathbf{y}$, as shown in Eq. (3.8). The derivative of $J(\theta)$ with respect to θ is shown in Eq. (3.9). The class probabilities, which are shown in Eq. (3.10), add up to one. The minimization of $J(\theta)$ is based on iterative optimization techniques [42] ,[43].

$$h_\theta(x^{(i)}) = \begin{bmatrix} P(\mathbf{y}=0|\mathbf{x}^{(i)}) \\ P(\mathbf{y}=1|\mathbf{x}^{(i)}) \\ P(\mathbf{y}=2|\mathbf{x}^{(i)}) \\ \ldots \\ P(\mathbf{y}=K|\mathbf{x}^{(i)}) \end{bmatrix} = \frac{1}{\sum_{j=1}^N e^{\theta(j)^T\mathbf{x}^{(\mathbf{i})}}} \begin{bmatrix} e^{\theta(0)^T\mathbf{x}^{(i)}} \\ e^{\theta(1)^T\mathbf{x}^{(i)}} \\ e^{\theta(2)^T\mathbf{x}^{(i)}} \\ \ldots \\ e^{\theta(N)^T\mathbf{x}^{(i)}} \end{bmatrix} \tag{3.7}$$

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^m \sum_{n=0}^N 1\left\{\mathbf{y}^{(i)} = n\right\} \log P(\mathbf{y}^{(i)} = n|\mathbf{x}^{(i)};\theta)) \tag{3.8}$$

$$\frac{\partial J(\theta)}{\partial \theta_{ln}} = -\sum_{j=1}^m x^{(i)}(1\{y^{(i)}=n\} - P(y^{(i)}=n|x^{(i)};\theta)) \tag{3.9}$$

$$P(\mathbf{y}^{(i)} = n|\mathbf{x}^{(i)};\theta)) = \frac{e^{\theta(k)^T\mathbf{x}^{(i)}}}{\sum_{l=1}^K e^{\theta(l)^T\mathbf{x}^{(i)}}} \tag{3.10}$$

3.2 Gradient descent optimization algorithms

Gradient descent is one of the most popular algorithms for solving machine learning optimization problems. The parameter θ in $J(\theta)$ is updated along the opposite direction of $\nabla_\theta J(\theta)$, as shown in Eq. (3.12). The learning rate parameter η controls the amount of change applied to θ in each optimization iteration [49], [44], [50]. The gradient computation in Eq. (3.11) is repeated for every input data vector (i.e. data sample).

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta^{(t)}; \mathbf{x}, \mathbf{y}) \tag{3.11}$$

- *Batch mode*: in batch mode, parameter update is defined by the error gradient average over the entire training data set. If the training data set is large, which is usually the case, parameter update takes a long time, if it is not infeasible in terms of memory consumption. The presence of similar vectors within the training data set renders batch mode computations highly redundant, which leads to the waste of computational resources;
- *Stochastic mode*: in contrast to batch mode, parameters are updated for every input vector $\mathbf{x}$ and respective target $\mathbf{y}$. Parameter update is thus very fast, and redundancy in gradient computation is reduced. However, updates are highly variant because gradients computed from a single sample are usually very different;
- *Mini-batch mode*: in mini-batch mode, parameter update is defined by the error gradient average over a subset of the training data set. The subset is usually small, with size ranging from around 50 to 256. The mini-batch mode error convergence is more stable than the stochastic mode error convergence, and the mini-batch parameter update is faster than the batch mode parameter update.

For large datasets, basic gradient descent algorithms may become inefficient and the training error may take a long time to converge. More sophisticated gradient descent algorithms have been widely used. To make gradient and parameter update expressions short, the assignment shown in Eq. (3.12) is considered in the following sections. The gradient computed at the current time step is g_t, and Eq. (3.12) is used next to explain some of the most popular gradient descent algorithms [49], [44], [50]:

$$\mathbf{g}_t \leftarrow \nabla_{\theta_t} J(\theta_t; \mathbf{x}_t, \mathbf{y}_t) \tag{3.12}$$

- *Momentum*: Eqs. (3.13) and (3.14) show the momentum update expressions, where γ is the momentum factor. It is usually set to a value around 0.9 [51], [49]. Using momentum attenuates oscillations in the training loss function curve (it further stabilizes gradient computations in the mini-batch mode), and it effectively changes the update size when many gradient computations yield parameter update along the same direction. The Nesterov momentum [52], which is a popular momentum update expression, is a variant of the basic momentum update expression;

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} - \eta \mathbf{g}_t \tag{3.13}$$

$$\theta_{t+1} = \theta_t + \mathbf{v}_t \tag{3.14}$$

- *Adagrad*: the Adagrad algorithm [27] scales η individually for each parameter, according to past gradient values, in order to speed up the converge of stochastic gradient descent algorithms. Eq. (3.15) defines $\mathbf{m}_t$ as an accumulator of past gradients, from which updates with different step sizes are computed for each parameter according to Eq. (3.16) [53], [27]. Details about the smoothing factor ϵ are provided in a following discussion about the RMSprop algorithm;

$$\mathbf{m}_t = \mathbf{m}_{t-1} + \mathbf{g}_t^2 \tag{3.15}$$

$$\theta_{t+1} = \theta_t + \frac{\eta}{\sqrt{\mathbf{m}_t} + \epsilon} \odot \mathbf{g}_t \tag{3.16}$$

- *Adadelta*: the Adadelta algorithm [18] is based on the Adagrad algorithm. It uses an adaptive learning rate that comprises both the magnitude of recently computed gradients and the magnitude of recent update steps. Adadelta addresses two drawbacks of Adagrad: the continuous decay of learning rates, and the requirement of a manually adjusted global learning rate. Error minimization remains effective after many updates have been done [54], [49];

- *RMSprop*: this algorithm corresponds to unpublished work, and it was introduced by Geoffrey Hinton in Lecture 6e of his Coursera class [28], [49]. It uses an adaptive learning rate, as in Adadelta. The $\mathbf{m}_t$ accumulator is shown in Eq. (3.17). It is computed using past gradients, as in Adagrad and in Adadelta, but RMSprop uses a exponentially weighted moving average of past gradients. Similarly to the momentum algorithm, $E[\mathbf{g}^2]_t$ is scaled by a momentum factor γ, to adjust the step size, and then subtracted by the scaled squared gradient. The parameter update rule is shown in Eq. (3.18). As the learning rate η is adjusted by $\mathbf{m}_t$, the RMSprop running average reduces abrupt variations that

may occur in the parameter update vector. The smoothing factor ϵ avoids division by zero, and it is usually in the range from 10^{-9} to 10^{-6} [28], [55], [49];

$$\mathbf{m}_t \leftarrow E[\mathbf{g}^2]_t = \gamma E[\mathbf{g}^2]_{t-1} - (1-\gamma)\mathbf{g}_t^2 \tag{3.17}$$

$$\theta_{t+1} = \theta_t + \frac{\eta}{\sqrt{\mathbf{m}_t} + \epsilon} \odot \mathbf{g}_t \tag{3.18}$$

- *Adam*: the adaptive moment estimation is one of the most popular optimization algorithms for neural network training, because of its relatively fast convergence to a local minimum close to the initialization point [29]. It combines the RMSprop algorithm and the momentum algorithm using an exponentially decaying average of past squared gradients. The gradient sequence first and second-order moment estimates, which are usually referred to as the mean $\mathbf{m}_t$ and the uncentered variance $\mathbf{v}_t$, are shown in Eqs. (3.19) and (3.20). Usual values for the β_1 and β_2 parameters are 0.9 and 0.999 [29], [49].

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} - (1-\beta_1)\mathbf{g}_t \tag{3.19}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \tag{3.20}$$

If β_1 and β_2 are close to 1, then $\mathbf{m}_t$ and $\mathbf{v}_t$ are biased toward zero, particularly in initial iterations. The computed bias-corrected moment estimates are shown in Eqs. (3.21) and (3.22). The parameter update expression is presented in Eq. (3.23) [29], [49].

$$\widehat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t} \tag{3.21}$$

$$\widehat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t} \tag{3.22}$$

$$\theta_{t+1} = \theta_t + \frac{\eta}{\sqrt{\widehat{\mathbf{v}}_t} + \epsilon} \odot \widehat{\mathbf{m}}_t \tag{3.23}$$

3.3 Multi-Layer Perceptron

This section briefly discusses the multi-layer perceptron (MLP), starting with the McCulloch-Pitts neuron and the perceptron in Section 3.3.1, and finishing with the MLP itself in Section 3.3.2.

3.3.1 Perceptron

A perceptron is a simple mathematical processing unit inspired by the biological neuron. Neural electrical signals are represented by numerical values. As these electrical signals are modulated by the strengths of synaptic connections between dendrites and axons, the perceptron computes a weighted sum of its input signals. The value of the weighted sum is usually limited by an activation function that is sigmoidal, or by a hyperbolic tangent function [56]. These operations are described in Eq. (3.24), including a bias parameter b and the non-linear activation function φ. This basic mathematical processing unit is also usually referred to as the McCulloch-Pitts neuron. Figure 3.3 shows a single neuron with three inputs.

$$y = \varphi(\sum_i w_i x_i + b) \tag{3.24}$$

A single neuron can only solve linearly separable binary classification problems. Common examples of linearly separable classification problems are the "AND" and "OR" logic functions. The McCulloch-Pitts neuron does not allow a solution for non-linearly separable classification problems, unless its inputs have been previously mapped into a representation in a different feature space [57]. A common example of non-linearly separable classification problem is the "XOR" logic function.

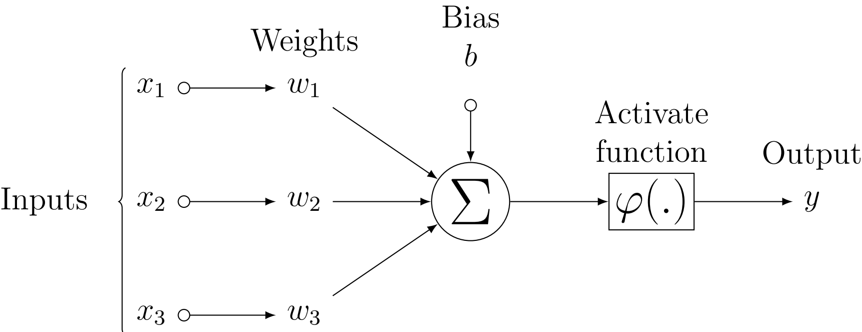


Figure 3.3: Artificial neuron.

3.3.2 Multilayer Perceptron

The multi-layer perceptron (MLP) is a feedforward neural network composed by layers of perceptrons. It solves non-linearly separable classification problems. The individual network nodes emulate biological neurons [58], and they are usually implemented by perceptrons or by McCulloch-Pitts neurons. The MLP layers are usually arranged along a forward direction, and so the network nodes form a graph with no cycles. Each layer is fully connected to the previous one (i.e. any network node receives input signals from all nodes in the previous layer). As weighted inputs

are successively mapped by non-linear activation functions in successively layers, we expect to obtain input representations that are increasingly better, in the sense of solving the classification or regression task at hand. To effectively find those representations, we optimize those weights for a specific problem using a training data set and an error backpropagation algorithm [43], [44], [57].

The MLP last layer may have a single output node (for regression or binary classification), or a number of output nodes equal to the number of classes (for classification). In the layers between the input layer and the output layer, which are also denoted as hidden layers, activation functions perform non-linear mapping, thereby transforming input data into linearly separable data (in the case of classification problems). Sigmoidal or hyperbolic tangent activation functions are commonly used. Neurons at the same layer usually have the same type of activation function [44], [57]. Other activation functions appear in the literature [59], [60], and Table 3.1 shows the activation functions that are used in the present work. A general MLP architecture is depicted in Figure 3.4.

Activation Functions			
Name	Math expression	Computational cost	Curves
Linear	x	1	
Sigmoidal	$\frac{1}{1+e^{-x}}$	$T(n)$	
Hyperbolic Tangent	$\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$2T(2n)$	
ReLU	$max(0,x)$	~ 1	

Table 3.1: Activation functions used in the present work.

Figure 3.5 illustrates a neural network solution obtained for a classification problem, considering a loss function such as the one in Eq. (3.8). A complex N-dimensional loss function, depicted by a one-variable function, is shown in the 2-D plot of Figure 3.5a. A particular set of neural network weights was updated three times, with respect to a single parameter θ (along the horizontal axis of Figure 3.5a), and the updated values are also shown in Figure 3.5a. The best fit corresponds to the red point, because that point corresponds to the smallest loss function value

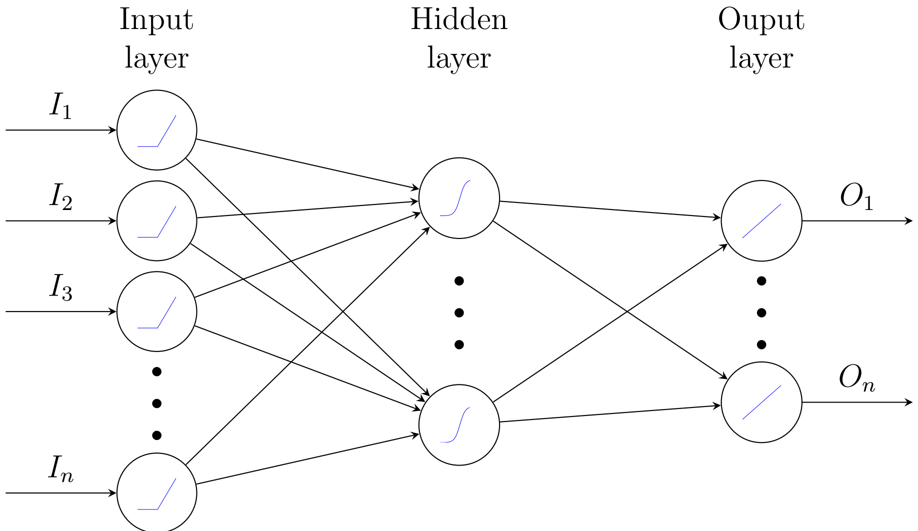
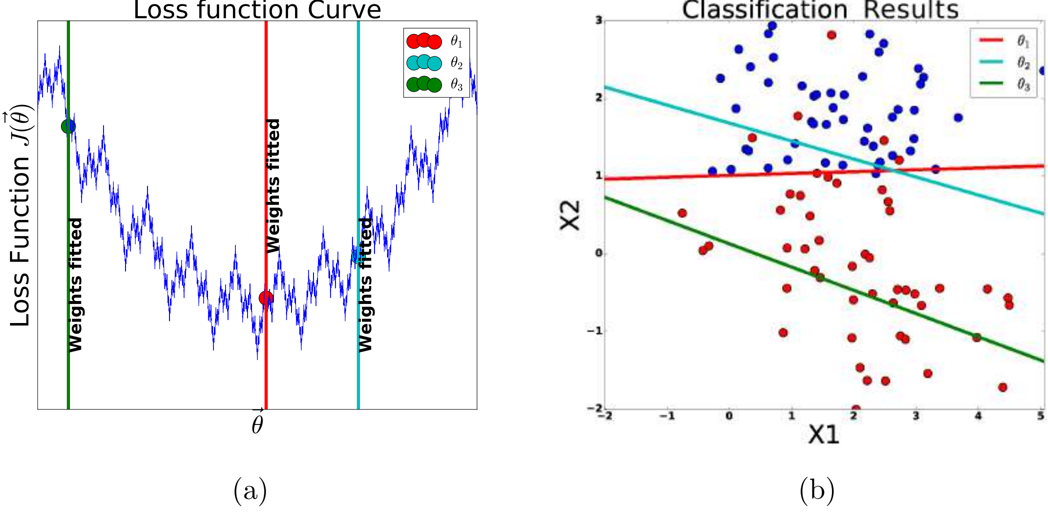


Figure 3.4: General architecture of a multi-layer neural network.

(among the red, green, and blue points), and it also corresponds to the smallest classification error in Figure 3.5b [61], [42], [44], [57].



(a) (b)

Figure 3.5: Neural network solution for a classification problem.

In recent works, more sophisticated neural networks architectures have been developed. Deep neural networks have a large number of hidden layers, and they lead to remarkable improvements in comparison to conventional MLP results. In this work, we will focus on deep convolutional neural networks.

3.4 Convolutional Neural Networks

Nowadays, the deep convolutional neural network [7], [62] is one of the most widely used machine learning techniques. It has been successfully applied to automatic

feature extraction for image analysis. Conventional or hand-crafted feature extraction is replaced by convolutional layers. The convolution operation is carried out by sliding a kernel on the image and, for every pixel on which the kernel is centered, computing the inner product between the kernel values and the underlying pixels. The first convolutional neural networks (CNNs) were based on the discrete convolution operation, and they were inspired by biological vision systems. Like the conventional CNNs, the deep CNNs have layers that are composed by two basic stages: the convolutional stage, which extracts features from the layer inputs, and the subsampling stage, which reduces image resolution [8], [57], [63], [64], [9].

3.4.1 Convolution Layers

Convolutional layers leverage the ideas of local connectivity, parameter sharing, and spatial arrangement [63], [8]. After training is complete, features extracted by convolutional layers usually yield better classification or regression overall performance than conventionally designed or hand-crafted features do.

- *Local connectivity*: each neuron in the convolutional layer is connected only to a small subset (a local neighborhood, defined by width, height, and depth) of the convolutional layer input image. The input image may be a real-world image, or a feature map created by a previous convolutional layer [8], [63]. Local connectivity makes training easier [8]. The use of kernels reduces the number of parameters in comparison to the number of parameters in a conventional (fully connected) MLP [63]. A reduced number of parameters is also useful to avoid overfitting. The Eq. (3.25) shows the convolution operation in convolutional layers for one feature map;

$$z_i = X \cdot W_i + b_i \tag{3.25}$$

- *Spatial arrangement*: in any convolutional layer, the neurons are arranged according to some CNN hyperparameters [1], in order to define the convolution operation output size. The hyperparameters considered in this case are input volume W, stride S, and padding P [8]. Depending on the number of the input image channels (i.e. the number of color fields), the first convolutional layer usually has a depth equal to one or equal to three. As we look into deeper layers, the input volume usually has depth larger than three. The stride parameter defines the step with which we will slide the current convolutional

[1] Hyperparameters are model properties (topology, for instance) and design technique properties (learning rate or momentum, for instance) that are adjusted by the designer during model development and training. The *hyper* prefix is used to distinguish those parameters (model or design properties) from the specific model parameters that are subject to optimization.

layer kernel over the input image. We usually have $S = 1$ or $S = 2$. The padding parameter is used for adjusting the input image resolution (by padding it with zeros along the vertical and horizontal directions), so that the ratio between the input image resolution and output image resolution is set to an integer value or to an equivalently simple ratio. To compute the size of an output volume O (i.e. the convolutional layer output size), we use Eq. (3.26), which involves the previously defined W, S, and P hyperparameters, and also the filter (kernel) size F [65];

$$O = \frac{W + 2P - F}{S} + 1 \tag{3.26}$$

- *Parameter sharing*: neurons are organized into feature maps, so that the weights (kernel values) connecting a local region (neighborhood) of the input image to an output neuron are the same for all output neurons. Since a single weight matrix (kernel) is used for extracting features for every valid pixel at the output image, the number of training parameters is clearly reduced. Indeed, the term *kernel* stems from the fact that the connection weight sets are repeated throughout all the convolutional layer neurons, and such sets thus behave as kernel filters. Figure 3.6 presents the weight connections as a $K \times K$ kernel connecting a $H_1 \times W_1 \times D_1$ (width, height, and depth) input volume to a $H_2 \times W_2 \times D_2$ output volume, this operation is performed by means of Eq. (3.25) and the final output is calculated through Eq. (3.26) [8] ,[65].

Every convolutional layer is completely described by a set of kernel filters. As the coefficients of each filter are optimized during deep CNN training, each filter learns how to extract specific patterns (features) from the layer input volume (i.e. multi-channel map). The convolution is a linear operation. After the convolution operation is complete, activation functions introduce non-linearity in the forward signal flow. Every output feature map is the result of a non-linear activation function (usually ReLU) applied to the convolution operation performed at the respective layer. After training is complete, extracted features become more complex (and more useful for solving the specific problem at hand) as we look into deeper layers.

3.4.2 Pooling Layers

The sub-sampling layer, or commonly called pooling layer, is used immediately after every convolutional layer, to reduce the resolution for every feature map. The pooling kernels resize their input volumes, which are convolutional layer output volumes, using mathematical or logic functions. The pooling kernels are usually

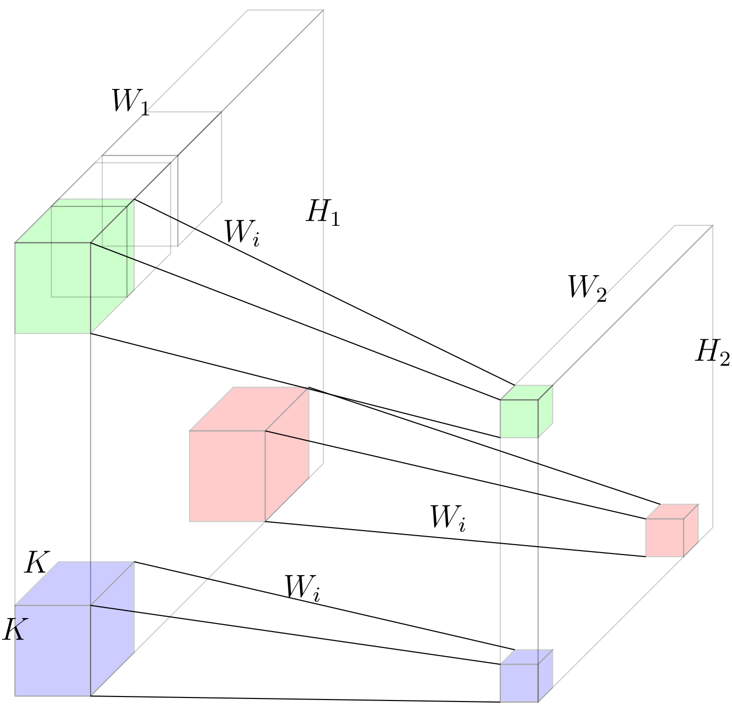


Figure 3.6: Connections between $H_1 \times W_1 \times D_1$ input volume and $H_2 \times W_2 \times D_2$ output volume.

square and their size is denoted as $K \times K$. They are applied with stride K over the feature map. The two most popular pooling kernels consist in selecting the maximum value within the $K \times K$ kernel area (max pooling) or in computing the average value within the kernel area (average pooling).

Using pooling layers has three advantages: the number of neural network parameters subject to optimization is significantly reduced, the network performance becomes more robust to outlying local variations that may occur within small neighborhoods at the input image, and, in deep CNNs with basic topology, the feed-forward computational cost is reduced (for layers in which the output volume is smaller than the input volume). The pooling layer output volume can be computed from Eq. (3.27), where W and S are the previously defined width and stride hyperparameters. The spatial extent hyperparameter F is analogous to the previously defined kernel size F, and so the same symbol is used for both hyperparameters. Depth remains unchanged, because pooling is only performed along the width and height dimensions of the input volume (feature map volume), regardless of its depth. Figure 3.7 illustrates a pooling operation over an $H_1 \times W_1 \times D_1$ input volume , using a $K \times K$ kernel. If $K = 2$, each 2×2 with stride $S = 2$ neighborhood is reduced to size 1×1. The width and height are thus reduced by a factor of two, and the depth remains equal, this means $H_2 = H_1/2$, $W_2 = W_1/2$ and $D_2 = D_1$ [8], [65], [8].

$$O = \frac{W - F}{S} + 1 \tag{3.27}$$

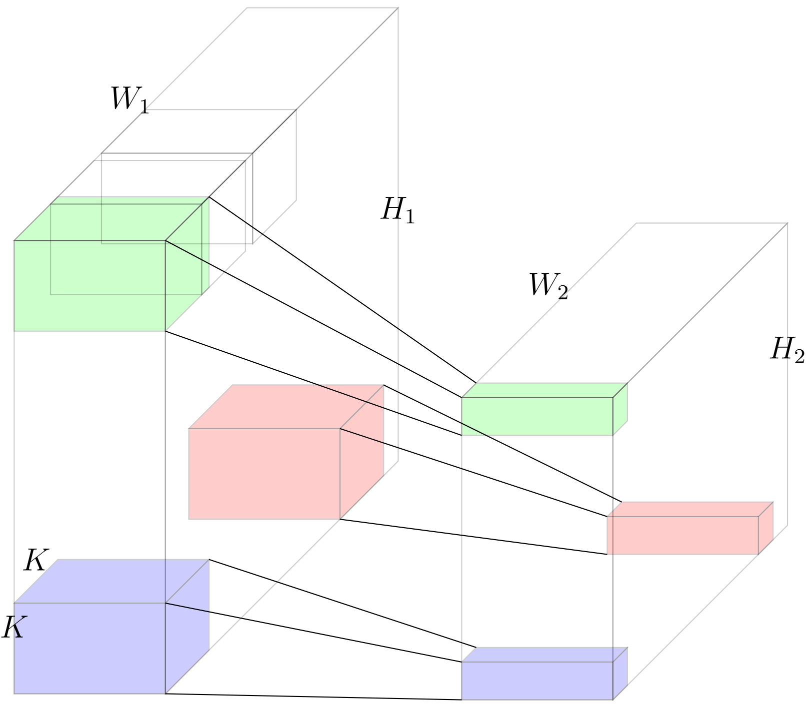


Figure 3.7: Connections between input and output volumes, illustrating downsampling by a factor of two at the pooling layer (i.e. at the convolutional layer output).

3.4.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are often composed by cascading feature-extracting stages containing a convolutional layer, an activation function, and a pooling layer. Each stage generates a feature map that is fed to the next feature-extracting stage. We think of this cascade of stages as a locally-connected feed-forward graph. After feature extraction is performed at the deepest convolutional/pooling stage, the respective feature maps are flattened into a $1 \times 1 \times D$ vector, where D is the dense neural network input dimension. The $1 \times 1 \times D$ vector is thus fed into the first fully-connected layer. The fully-connected layers are equal to the layers in a conventional MLP. The fully-connected layers are also referred to as *dense* layers [66]. Figure 3.8 depicts a typical CNN topology (which is not among our final ones) composed by two convolutional layers, two respective (max) pooling layers, one additional (average) pooling layer, and three dense layers [8], [63], [65]. This topology uses 16 filters at the first convolutional layer, and 32 filters at the second one. After both convolutional layers, max-pooling layers with 2×2 kernels are commonly used. The third pooling layer (average pooling) further reduces the number of dimensions at the MLP input, in order to reduce overfitting. The last

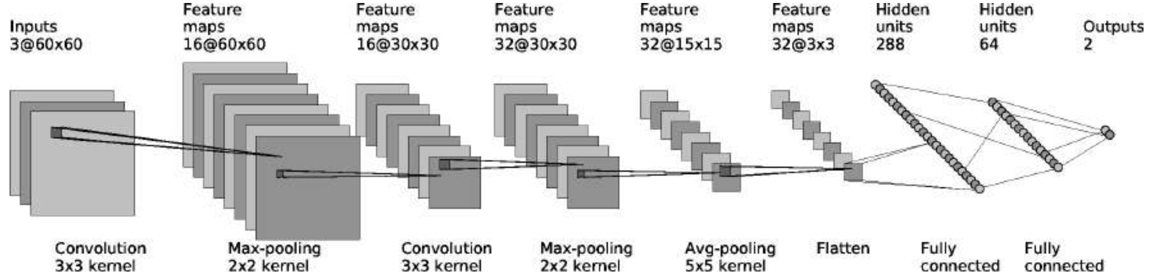


Figure 3.8: CNN topology using two stages (convolutional layer and pooling layer) repeated twice, thus composing two CNN layers, immediately followed by an additional pooling layer, flattening, and three fully-connected layers.

feature maps (32 3×3 feature maps) are flattened and fed into the MLP. The first two layers of the MLP map the extracted features into data that are more representative for classification. The final classification is handled by the output dense layer, which is usually composed by maximally sparse neurons (i.e. one neuron for each class) [44]. To compute class probabilities, we apply the softmax operation (Eq. (3.10)) to the neural network final outputs. Sometimes, the softmax operation is regarded as an additional layer that generates strongest output at the output node corresponding to the predicted class [65].

3.5 Regularization

One challenge in machine learning consists in keeping, in a test data set that contains previously unseen input data, classification or regression accuracy close to the accuracy obtained upon training interruption. Deep models are usually more powerful than shallower or simpler models, but the deep models tend to have overfitting problems, because of the large number of parameters to be optimized. Many strategies for obtaining deep neural network models with a good generalization performance, i.e. without overfitting, have been developed. Many of those strategies are based on using penalty terms or weight decay in order to limit model size, parameter values, and effective training extension. Effective regularization techniques reduce variance without increasing model bias. In neural network applications, regularization techniques penalize the loss function over the training data or perform a limitation over the number of parameters to be optimized. If the bias was regularized as well, then underfitting problems might occur [8], [57]. Some of the currently available regularization strategies are explained next.

- L^1 *regularization*: this method uses a parameter norm penalty term whose expression is shown in Eq. (3.28). The $\Omega(\theta)$ term is weighted by a scaling factor, and it is then added to the original loss function, which yields the regularized loss function that is shown in Eq. (3.29). In Eq. (3.29), the original

loss function is $J_0(\theta, \mathbf{x}, \mathbf{y})$, and the scaled regularization term is $\alpha\Omega(\theta)$ [57], [43], [8]. In the sense of limiting synaptic connection strength, the L^1 regularization technique is related to the weight decay regularization technique. The L^1 regularization is computationally inefficient if it is applied to non-sparse input data, and it may lead to sparse outputs;

$$\Omega(\theta) = \| \theta \|_1 = \sum_j | \theta_j | \tag{3.28}$$

$$J(\theta; X, y) = J_0(\theta; X, y) + \alpha\Omega(\theta) \tag{3.29}$$

- *L^2 regularization*: this is one of the most common regularization techniques used in machine learning. The L^2 regularization penalizes the squared magnitude of the parameters. The penalty term expression is shown in Eq. (3.30). In contrast to L^1 regularization, the L^2 regularization is computationally efficient and does not lead to sparse outputs. The regularized loss function can be expressed as in Eq. (3.29). Combination of L^1 and L^2 regularization is mentioned in [67] , and it is also known as *elastic net regularization* [8], [43], [47], [48];

$$\Omega(\theta) = \| \theta \|_2^2 = \sum_j \theta_j^2 \tag{3.30}$$

- *Dropout*: this regularization technique is extremely effective, simple, and computationally cheap [25]. Dropout overcomes L^1, L^2, and other regularization techniques. It works through temporary modification of the neural network structure itself. To implement dropout, we suppress the outputs of a random subset of the neural network neurons, for every gradient computation. After parameter update (only for the neurons that remained active), the previously suppressed neurons are restored, and a new neuron subset is randomly selected for dropout. This process is repeated until all neurons have been suppressed at least once. Dropout configuration involves only one hyperparameter, which defines the dropout keep probability, ranging from 0.5 to 0.9 depending on the model complexity [25], [57], [8];

- *Batch normalization (as a regularizer)*: batch normalization was introduced in [26]. It potentially leads to higher overall accuracy and to faster learning, by adjusting input data distributions around zero mean and unit variance for all neural network layers. Zero mean and unit variance input normalizations may be undone by batchnorm parameter optimization, if training data deems

it effective for reducing internal covariate shift[2] problems. In other words, the neural network can learn how to undo the zero mean and unit variance input normalization at specific layers, if that is needed for reducing internal covariate shift. Batch normalization adds approximately 30% computational overall overhead to the feed-forward and parameter update iterations [68], but it leads to conveniently normalized data at the inputs for all layers in the network. This reduces internal covariate shift, and regularize the gradient from distraction to outliers among the input data samples and flow towards the local minimum, accelerating the learning process [26], [8]. Batch normalization also works as a regularizer, because the normalization of neural network layer inputs according to batch statistics adds noise to the parameter update operations. The same data sample affects normalization differently, if it is present in different mini-batches;

- *Global Average Pooling*: in global average pooling (GAP) [66], the spatial average of each feature map (at the last convolutional layer) is computed, and the resulting vector is fed to the classification (dense) layers. In [66], the authors explain why performing global average pooling may lead to better results than feeding the last convolutional layer feature maps directly to the dense layers. The regularization associated with GAP acts as a structural and non-parametric regularizer. GAP is not strictly a regularization technique, but it may be regarded as an aid to regularization, as its application reduces the number of dense layer parameters. Techniques similar to GAP have been developed. For example, in global max pooling [69], the averaging operation is replaced by the max operation;

- *Data Augmentation*: if the number of samples in a data set tends to infinity, then a sufficiently large model will learn the data distribution perfectly and present optimal test performance. Thus, augmenting (artificially increasing the number of samples in) the training data set by applying controlled transformations to the original data samples may improve learning. For image databases, popular operations include random crops, flipping, rotation, color domain modifications, color jittering, as well as different combinations of these operations. Data augmentation is widely used in deep learning, especially if the training database size is not large [47] ,[66], [70].

[2]Internal covariate shift is the process through which the inputs of any neural network layer define a nonstationary probability density function. Throughout training, the probability density function associated with the inputs of a given neural network layer changes, as the parameters in previous layers are updated.

3.6 Backpropagation Algorithm

Backpropagation [71] is a widely used, computationally simple, algorithm for computing gradients for parameter update [8] in deep feedforward networks. The error is estimated by the loss function $J(\theta)$, which indicates the average distance (in some arbitrary sense) between the targets and the predicted outputs [44], [61], [8], [72]. The backpropagation algorithm yields the gradient $\nabla_\theta J(\theta)$ for almost any loss function J, where θ is a multidimensional parameter to be optimized, even if the number of parameters is larger then the number of input arguments in J (i.e. the number of input dimensions in the neural network model). To compute the gradient at any given layer out of previously computed gradients (which are already available for all layers that closer to the output than the given layer), backpropagation uses the chain rule. A clear derivation of general expressions for backpropagation may be obtained from graph theory [8].

The backpropagation algorithm has two stages: forward propagation and backward propagation. At the forward propagation stage, a data sample is fed to the neural network input, and information flows throughout the network towards its output. At the end of this stage, the loss function between the network output and the input data sample target is evaluated (partially, in the mini-batch mode, or fully, in the full batch mode) [72], [61]. At the backward propagation stage, loss gradients are successively computed for parameters at every layer, as the error signals are propagated from the network output back towards its input [8], [50].

Algorithms 1 and 2 describe the general backpropagation operations that are used for deep neural network training, in the forward and backward propagation stages, respectively [8]. The model parameters θ are presented as weight and bias terms in neural networks represented as $\mathbf{W}$ and $\mathbf{b}$ respectively, the pre-activation function is $\mathbf{z}$, the arbitrary activation function is $\mathbf{f}$, the hidden layer outputs are $\mathbf{h}$, and the computed gradients are stored in $\mathbf{g}$ [8]. At the forward stage, the neural network acts as a forward graph connecting inputs to outputs. At the backward propagation, a backward graph connects the neural network outputs to its inputs, and gradients values are available at each node of the backward graph. Algorithm 2 starts at the neural network output, by computing loss function derivatives with respect to the output layer parameters. These loss function derivatives are subsequently propagated to previous (hidden) layers, through the hidden neurons and their respective activation functions.

Algorithm 1 : Forward computation in a generic deep feedforward neural network.

Input: ℓ, Network depth

Input: $\mathbf{W}^{(i)}$, $i \in \{1...\ell\}$, model weight matrices of the model

Input: $\mathbf{b}^{(i)}$, $i \in \{1...\ell\}$, bias parameters of the model

Input: $\mathbf{x}$, model inputs

Input: $\mathbf{y}$, targets

1: $\mathbf{h}^{(0)} = \mathbf{x}$ # Neural network inputs
2: **for** k = 1, ... , ℓ **do**
3: $\mathbf{z}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}$
4: $\mathbf{h}^{(k)} = \mathbf{f}(\mathbf{z}^{(k)})$
5: **end for**
6: $\hat{\mathbf{y}} = \mathbf{h}^{(\ell)}$
7: $\mathbf{J} = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda\Omega(\theta)$ # Loss function plus regularization term

Algorithm 2 : Backward computation in a generic deep feedforward network.

Input: ℓ, Network depth

Input: L, loss function of the model

1: $\mathbf{g} \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{\mathbf{y}}, y)$ # Loss function gradient
2: **for** k = ℓ, $\ell - 1$, ... , 1 **do**
3: $\mathbf{g} \leftarrow \nabla_{\mathbf{z}^{(k)}} J = \mathbf{g} \odot \mathbf{f}'(\mathbf{z}^{(k)})$
4: $\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega(\theta)$
5: $\nabla_{\mathbf{W}^{(k)}} J = \mathbf{g}\ \mathbf{h}^{(k-1)T} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega(\theta)$
6: $\mathbf{g} \leftarrow \nabla_{h^{(k-1)}} J = \mathbf{W}^{(k)T}\mathbf{g}$ # Propagate the gradient in all lower layers
7: **end for**

Specifically, CNNs have convolutional, pooling, and dense layers. The backpropagation algorithm operations for deep CNNs are described in Algorithms 3 and 4, for the forward and backward propagation stages, respectively [8]. In 3, an input image is fed to the convolutional layers. All convolutional layers generate feature maps, and their feature maps are immediately resized at the respective pooling layers. Activation functions are used in lines 4 and 5 of 3.

Algorithm 3 : Forward computation in convolutional and pooling layers.

Input: ℓ, Network depth

Input: $\mathbf{W}^{(i)}$, $i \in \{1...\ell\}$, model weight matrices of the model

Input: $\mathbf{b}^{(i)}$, $i \in \{1...\ell\}$, bias parameters of the model

Input: $\mathbf{X}$, model inputs

Input: $\mathbf{h}$, flatten output

1: $\mathbf{h}_N^{(0)} = \mathbf{X}$ # N could be 1 or 3 for gray or color channels
2: **for** k = 1, ... , ℓ **do**
3: $\mathbf{z}_N^{(k)} = \mathbf{W}_{N_{x,y}}^{(k)} * \mathbf{h}_{N_{x,y}}^{(k)} + \mathbf{b}_{N_{x,y}}^{(k)}$
4: $\mathbf{p}_N^{(k)} = \mathbf{Pooling}(\mathbf{f}(\mathbf{z}_N^{(k)}))$
5: $\mathbf{h}_N^{(k)} = \mathbf{p}_N^{(k)}$
6: **end for**
7: $\hat{\mathbf{h}}^{(\ell)} = \mathbf{Flatten}(\mathbf{h}_N^{(\ell)})_{Nx1}$ # Reshape the matrix into vector of neurons

In Algorithm 4, CNN backpropagation starts at the input of the first dense layer, assuming that regular backpropagation was already applied from the overall neural network output back to the first dense layer input. The first operation to be considered is pooling. As this operation involves functions that do not depend on weights, the pooling layer is traversed without any weight update, which obviously also reduces backpropagation computational cost. If max-pooling is used, then the error backpropagation is directed toward the path where it came from. Other units in this pooling layer are not affected by the error. If average pooling is used, then the error is backpropagated with scaling factor equal to $1/K^2$, where K is the kernel size, and it is assigned to all units that compose the kernel. Algorithm 5 describes parameter update in batch mode, which is performed after loss function gradients have been computed for all layers.

Algorithm 4 : Backward computation in convolutional and pooling layers.

Input: ℓ, Network depth

Input: $\nabla_{h^{(\ell+1)}} J \leftarrow \mathbf{W}^{(\ell+1)T}\mathbf{g}$, gradient of first hidden fully connected layer

1: $\mathbf{g} \leftarrow \nabla_{h^{(\ell+1)}} J$
2: **for** k = ℓ, $\ell - 1$, ... , 1 **do**
3: $\mathbf{p}_N \leftarrow \mathbf{Upsampler}(\mathbf{g})_N$ # there is no direct effect for weights
4: $\mathbf{g}_N \leftarrow \nabla_{\mathbf{z}_N^{(k)}} J = \mathbf{p}_N \odot \mathbf{f}'(\mathbf{z}^{(k)})$
5: $\nabla_{\mathbf{b}_N^{(k)}} J = \mathbf{g}_N + \lambda \nabla_{\mathbf{b}_N^{(k)}} \Omega(\theta)$
6: $\nabla_{\mathbf{W}_N^{(k)}} J = \mathbf{h}_N^{(k-1)T} * rot_{180}(\mathbf{g}_N) + \lambda \nabla_{\mathbf{W}_N^{(k)}} \Omega(\theta)$
7: $\mathbf{g}_N \leftarrow \nabla_{h^{(k-1)}} J = \mathbf{W}_N^{(k)T}\ \mathbf{g}_N$
8: **end for**

Algorithm 5 : Weights and bias update by mean of Backpropagation using batch mode.

Input: ℓ, Network depth

Input: L, loss function of the model

Input: $\mathbf{W}$,$\mathbf{b}$, Weights and bias of the model previously initialized

Input: x,y, inputs and target of the problem to solve previously initialized

1: **for** k = 1, ... , ℓ **do**

2: $\nabla_{\mathbf{W}^{(k)}} J$, $\nabla_{\mathbf{b}^{(k)}} J$ = **Backpropagation(L;W,b;x,y)**

3: $\mathbf{W}^{(k)} = \mathbf{W}^{(k)} - \alpha \frac{1}{\ell} \nabla_{\mathbf{W}^{(k)}} J$

4: $\mathbf{b}^{(k)} = \mathbf{b}^{(k)} - \alpha \frac{1}{\ell} \nabla_{\mathbf{b}^{(k)}} J$

5: **end for**

3.7 Initialization

Nowadays, initialization in neural networks is an important topic [68], there are many techniques to initialize weights and bias [68], [73], [74]. Uniform and Xavier initialization were used in this work.

- *Uniform*: This initialization operates in a uniform distribution to obtain the weights values. The intervals are set by designing criteria [68], [52];
- *Xavier*: This initialization process calculates the Eq. (3.31), where n_{in} and n_{out} are the fan in and fan out connections, i.e. the number of inputs and outputs of this neuron.

$$Var(W) = \frac{2}{n_{in} + n_{out}} \tag{3.31}$$

Chapter 4

Methodology

This chapter introduces the employed methodology that was employed to design the system taking into consideration the building input data based on feature extraction and deep learning architectures, including image enhancement and post-processing algorithm. The first section describes the feature extraction algorithms employed in the multi-layer-perceptron-based (**MLP-based**) system. The next section presents a detailed experimental analysis, comprising a discussion and comparisons about classifier architecture designs employed in this work: the MLP-based and the Convolutional-Neural-Network-based (**CNN-based**) systems. Later, a post-processing algorithm is presented, which is employed in order to reduce the false positive rate in the system. Spatial and temporal analysis are applied, and structural and procedure details are described such as to obtain favorable results.

4.1 Features Extractor Algorithms

The MLP network was applied to the database with six different feature extraction approaches, based on wavelet statistics, LBP, Hu moments, entropy and GLCM. The contrast, dissimilarity, homogeneity, energy and correlation of the pattern can be extract from the GLCM matrix [75]. In some networks, pre-processing was employed before the feature extraction procedure in order to remove artifacts. The corresponding features and pre-processing approach of each MLP topology are:

- *MLP 1*: To obtain a zero-mean grayscale window, the window mean grayscale value was subtracted from each pixel. A three-level dyadic wavelet decomposition was subsequently applied. Each 2-D wavelet decomposition level generates four decomposition subbands. Overall mean and variance were computed for each subband [11], leading to eight features per wavelet level. At the top level, the low-frequency subband mean value is zero, and so only its variance was computed. This leads to seven features at the top level, and thus 23 features

are used as MLP inputs. This descriptor is detailed in Algorithm 6. In line 5, the sequence filters are applied to the image in order to obtain the subbands $S_{n,1}$, $S_{n,2}$, $S_{n,3}$ and $S_{n,4}$ explained in the Figure 2.1a. Line 8 and 11 compute the wavelets from S_1 and $S_{1,1}$ in the same way as illustrated in Figure 2.1b;

- *MLP 2*: A three-level wavelet packet decomposition was applied to the zero-mean grayscale image. The overall mean and variance were computed for each subband, which means eight features from the first level, 32 features from the second level, and 128 features from the third level. Redundant information appears across different subbands. To remove redundant features, a correlation matrix was computed, and features whose correlation with a previously selected feature was above 0.5 were discarded [11], [13]. This procedure reduced the feature vector size to 25. This descriptor is detailed in Algorithm 7 and employs the wavelet block of Figure 2.1b;

- *MLP 3*: To compute LBP features, we selected 24 pixels from a circularly symmetric neighborhood with a three-pixel radius, and then we applied a uniform computation method on them [10], [37]. The number of uniform prototypes in an LBP depends on the number of pixels selected for uniform computation. The final histogram dimension number is equal to the number of selected points plus two [10], which leads to a 26-element feature vector;

- *MLP 4*: The LBP was computed for red, blue and green channels of every input window using the same parameters as in MLP 3. The average of the red, blue and green LBP histograms then composed the first 26 features that were used as inputs for MLP 4. In addition to those 26 features, we computed five GLCM coefficients from every input window graylevel representation, and we also computed maximum values from the hue and saturation histograms of the HSV representation of the input window. The overall size of the MLP 4 input feature vector is thus equal to 33. This descriptor is presented in Algorithm 8;

- *MLP 5*: Histogram equalization was applied to the saturation and brightness (value) components in the HSV representation of the input window. The window was then converted into a single grayscale channel, and the 26 LBP histogram and five GLCM features of the grayscale representation were calculated (31 features). The maximum value of the hue channel histogram and the entropy values of the red, green, blue, hue and saturation channels [75] were then included in the feature vector, which increased the feature vector size to 37. Finally, we computed a Canny-filtered version of the input window, and, from that filtered image, we computed seven Hu moments and five GLCM fea-

tures [40], which led to a feature vector with size equal to 49. This descriptor is presented in Algorithm 9;

- *MLP 6*: The same histogram equalization and grayscale conversion of MLP 5 were applied, and we computed the same 26 LBP features and five GLCM features of MLP 5. We then included, in the feature vector, the maximum values of hue, saturation, red and blue channel histograms, as well as the entropy values of the red, green, and blue channels, thus increasing the number of features to 38. Finally, we computed the first Hu moment (from the Grayscale channel) [40], five GLCM features from the hue channel, and five GLCM features from the saturation channel, which lead to a feature vector with size equal to 49. This descriptor is summarized in Algorithm 10.

Algorithm 6 : Feature extraction based on db2 wavelets.

Input: Image, input Image from dataset or extracted Window
Input: H_i, i=1,2 db2 wavelet filters
Output: f, features extracted from input image
1: $I \leftarrow GrayImage = \mathbf{RGBtoGRAY}(Image)$
2: I = $I - \mu_I$ # Subtract the mean
3: Sequence = [1,1 ; 1,2 ; 2,1 ; 2,2]
4: **for** n = 1, ... , 4 **do**
5: $F_n \leftarrow S_n = I * H_{Sequence(n)}$ # Image filtering
6: **end for**
7: **for** n = 1, ... , 4 **do**
8: $F_{4+n} \leftarrow S_{1,n} = S_1 * H_{Sequence(n)}$ # Image decomposition in next subband
9: **end for**
10: **for** n = 1, ... , 4 **do**
11: $F_{8+n} \leftarrow S_{1,1,n} = S_{1,1} * H_{Sequence(n)}$ # Image decomposition in next subband
12: **end for**
13: f = σ_{F_1} # Compute standard deviation from subband 1
14: **for** n = 1, ... , 11 **do**
15: $f_{2n} = \mu_{F_n}$ # Extract mean from each subband image
16: $f_{2n+1} = \sigma_{F_n}$ # Extract standard deviation from each subband
17: **end for**

4.2 Neural Networks System

This section describes the complete system and how to tackle the classification problem by means of the two machine learning techniques explained in Chapter 4, the MLP-based and CNN-based classifiers, in order to choose the best architecture and analyze the complete system including false positive reduction post-processing algorithms.

Algorithm 7 : Feature extraction based on least correlated db2 wavelets features.

Input: Image, input Image from dataset or extracted Window
Input: H_i, i=1,2 db2 wavelet filters
Output: f, features extracted from input image

1: $I \leftarrow GrayImage = \textbf{RGBtoGRAY}(Image)$
2: I = $I - \mu_I$ # Subtract the mean
3: Sequence = [1,1 ; 1,2 ; 2,1 ; 2,2] # Wavelet filter sequence
4: **for** $k = 1, ..., 20$ **do**
5: **if mod**(k-1,5)=0 **then**
6: $F_k \leftarrow S_k = I * H_{Sequence(k)}$ # Compute first-level subband images
7: **else**
8: $F_k \leftarrow S_{k,m} * H_{Sequence(m)}$ # Compute second-level subband images
9: **end if**
10: **end for**
11: **for** n = 1, ... , 20 **do**
12: $\mu_n, \sigma_n = \textbf{MeanVariance}(F_n)$ # Compute Mean and Variance
13: **end for**
selected features with the low correlated values computed for μ_n, σ_n using
function **ComputeCorrelation**(μ_n, σ_n)
14: f = $[\sigma_1, \mu_2, \sigma_2, \mu_3, \sigma_3, \mu_4, \sigma_4, \mu_5, \sigma_5, \mu_6, \sigma_6, \mu_7, \sigma_7, \mu_8, \sigma_8, \mu_{10}, \sigma_{10}, \mu_{12}, \sigma_{12}, \mu_{15}, \sigma_{15}, \mu_{16}, \sigma_{16}, \mu_{20}, \sigma_{20}]$

Algorithm 8 : Feature extraction based on color texture information.

Input: Image, input Image from dataset or extracted Window
Output: f, features extracted from input image

1: $I \leftarrow GrayImage = \textbf{RGBtoGRAY}(Image)$
2: $I_r, I_g, I_b, \leftarrow \textbf{ExtractChannels}(Image)$
3: $LBP_r = \textbf{LBP}(I_r, 24, 3,' uniform')$ # Compute red channel LBP
4: $LBP_g = \textbf{LBP}(I_g, 24, 3,' uniform')$ # Compute green channel LBP
5: $LBP_b = \textbf{LBP}(I_b, 24, 3,' uniform')$ # Compute blue channel LBP
6: $f_{LBP} \leftarrow \textbf{MeanVector}(LBP_r, LBP_g, LBP_b)$ # Compute the mean color LBP
7: $f_{GLCM} \leftarrow \textbf{GLCM}(I)$
8: $J \leftarrow HSVImage = \textbf{RGBtoHSV}(Image)$
9: $f_H \leftarrow \textbf{Max}(\textbf{histogram}(J_H, bins = 72))$ # Compute 72 bins for Hue channel
10: $f_S \leftarrow \textbf{Max}(\textbf{histogram}(J_S, bins = 20))$ # Compute 20 bins for Saturation channel

4.2.1 MLP-based System

The Algae detection system using MLP can be implemented by two ways: using pixels directly to feed the network or through feature extraction algorithms. The design of the classifiers takes into consideration regularization techniques, activation functions, batch size and influence of descriptors in performance. Additionally, the MLP-based system employs a false positive reduction procedure at the end, aiming to reduce the wrong classification in the system. Therefore, the complete MLP-based system is composed of three blocks as illustrated in Figure 4.1, where the **feature extractor block** is one of the algorithms explained in this chapter, the **neural network block** based on MLP will be fed by the feature vector to solve a 2-classes classification problem, and the false-positive reduction block will perform

Algorithm 9 : Feature extraction based on color, shape and texture information.

Input: Image, input Image from dataset or extracted Window

Output: f, features extracted from input image

1: $I_n \leftarrow Normed = \textbf{NormalizeImage}(Image)$ # Normalize values from 0 to 255
2: $J \leftarrow HSVImage = \textbf{RGBtoHSV}(I_n)$
3: $J_S = \textbf{HistEqualization}(J_S)$ # Histogram equalization in Saturation channel
4: $J_V = \textbf{HistEqualization}(J_V)$ # Histogram equalization in Brightness channel
5: $I_{pre} \leftarrow RGBImage = \textbf{HSVtoRGB}(J)$ # Pre-processed Image
6: $I \leftarrow GrayImage = \textbf{RGBtoGRAY}(I_{pre})$
7: $I_r, I_g, I_b, \leftarrow \textbf{ExtractChannels}(Image)$
8: $f_{LBP} \leftarrow LBP = \textbf{LBP}(I, 24, 3,'uniform')$ # Compute the LBP features
9: $f_{GLCM} \leftarrow \textbf{GLCM}(I)$ # Compute the GLCM features from gray image
10: $f_H \leftarrow \textbf{Max}(\textbf{histogram}(J_H, bins = 72))$ # Compute the Max bin from 72 bins for Hue channel
11: $E_r = \textbf{Entropy}(I_r,'rows')$ # Compute rows entropy in Red channel
12: $E_g = \textbf{Entropy}(I_g,'rows')$ # Compute rows entropy in Green channel
13: $E_b = \textbf{Entropy}(I_b,'rows')$ # Compute rows entropy in Blue channel
14: $E_H = \textbf{Entropy}(J_H,'rows')$ # Compute rows entropy in Hue channel
15: $E_S = \textbf{Entropy}(J_S,'rows')$ # Compute rows entropy in Saturation channel
16: $f_{Entropy} \leftarrow [E_r, E_g, E_b, E_H, E_S]$
17: $I_{blur} = \textbf{MedianBlur}(I)$
18: $I_{edge} = \textbf{Canny}(I)$ # Compute Canny edge detector
19: $f_{Hu} \leftarrow \textbf{HuMoments}(I_{edge})$ # Compute the Hu moments
20: $f_{GLCM_{edge}} \leftarrow \textbf{GLCM}(I_{edge})$ # Compute the GLCM features from Canny image

spatial and temporal analysis of the classifier result.

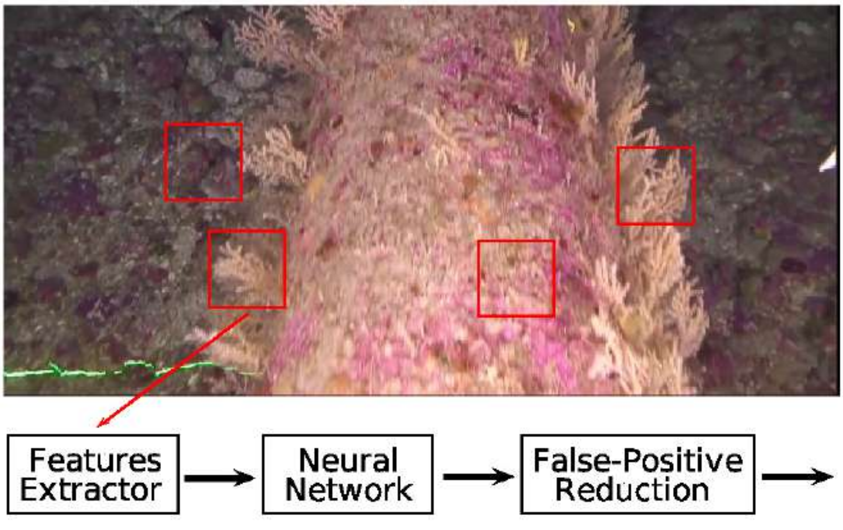


Figure 4.1: MLP-based system.

Experiments contemplates cross-validation, different feature extractors, initialization techniques and regularization methods. A testing set provided by 20% of the database images is employed to evaluate the performance of all MLP models.

Algorithm 10 : Feature extraction global and local based on color, shape and texture information.

Input: Image, input Image from dataset or extracted Window
Output: f, features extracted from input image
1: $I_n \leftarrow Normed = \textbf{NormalizeImage}(Image)$ # Normalize values from 0 to 255
2: $J \leftarrow HSVImage = \textbf{RGBtoHSV}(I_n)$
3: $J_S = \textbf{HistEqualization}(J_S)$ # Histogram equalization in Saturation channel
4: $J_V = \textbf{HistEqualization}(J_V)$ # Histogram equalization in Brightness channel
5: $I_{pre} \leftarrow RGBImage = \textbf{HSVtoRGB}(J)$ # Pre-processed Image
6: $I \leftarrow GrayImage = \textbf{RGBtoGRAY}(I_{pre})$
7: $I_r, I_g, I_b, \leftarrow \textbf{ExtractChannels}(Image)$
8: $f_{LBP} \leftarrow LBP = \textbf{LBP}(I, 24, 3,'uniform')$ # Compute the LBP features
9: $f_{GLCM} \leftarrow \textbf{GLCM}(I)$ # Compute the GLCM features from gray image
10: $f_{GLCM_H} \leftarrow \textbf{GLCM}(J_H)$ # Compute the GLCM features from Hue channel
11: $f_{GLCM_S} \leftarrow \textbf{GLCM}(J_S)$ # Compute the GLCM features from Saturation channel
12: $f_H \leftarrow \textbf{Max}(\textbf{histogram}(J_H, bins = 72))$ # Compute the Max bin from 72 bins for Hue channel
13: $f_S \leftarrow \textbf{Max}(\textbf{histogram}(J_S, bins = 50))$ # Compute the Max bin from 50 bins for Hue channel
14: $f_S \leftarrow \textbf{Max}(\textbf{histogram}(I_r, bins = 64))$ # Compute the Max bin from 64 bins for Red channel
15: $f_S \leftarrow \textbf{Max}(\textbf{histogram}(I_b, bins = 64))$ # Compute the Max bin from 64 bins for Blue channel
16: $E_r = \textbf{Entropy}(I_r,'rows')$ # Compute rows entropy in Red channel
17: $E_g = \textbf{Entropy}(I_g,'rows')$ # Compute rows entropy in Green channel
18: $E_b = \textbf{Entropy}(I_b,'rows')$ # Compute rows entropy in Blue channel
19: $f_{Entropy} \leftarrow [E_r, E_g, E_b]$
20: $I_{blur} = \textbf{MedianBlur}(I)$
21: $I_{edge} = \textbf{Canny}(I)$ # Compute Canny edge detector
22: $f_{Hu_1} \leftarrow \textbf{HuMoments}(I_{edge})$ # Compute the first Hu moment
23: $f_{GLCM_{edge}} \leftarrow \textbf{GLCM}(I_{edge})$ # Compute the GLCM features from Canny image

4.2.2 CNN-based System

Deep learning implementation based on CNN was also employed in the experiments to perform the classification task. The raw images came from the database (detailed in Appendix A), whose sizes were 60x60x3. These images are fed to the input convolutional layer, that passes through the net until the output layer. Other versions comprise CNN+MLP, before linking to the output layer.

The system design consists of initialization procedure, regularization techniques, activation functions, batch size selection and network configuration, the latter comprising the choices of the number of layers, pooling kernel sizes, filter-widths and patch size. Cross-validation was employed in order to give robustness to performance for all provide models [20]. A testing set provided by the 20% of database is employed to measure the performance for all CNN and CNN+MLP models. Comparison and discussion of the criteria employed to choose between CNN and CNN+MLP models will be described in the next chapter.

The CNN-based system is illustrated in Figure 4.2, using a configuration similar to the MLP-based system. The false positive reduction block was also included at

the output of the system, and convolutional layers were employed instead of the feature extractor block.

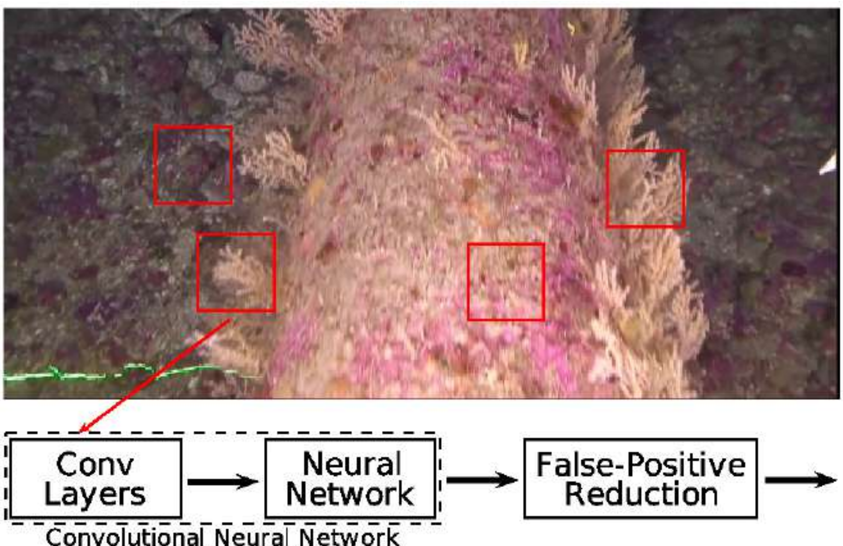


Figure 4.2: CNN-based System.

4.3 False Positives Reduction

As mentioned in a previous chapter, avoid false positives are more important than avoid false negatives in real scenes, given the nature of the problem. Post-processing algorithms, contained in the false-positive-reduction block, were applied to reduce false positive classifications produced by large shift steps performed by the window. The spatial and temporal analyses, which comprise the proposed post-processing algorithm, are described next.

Experiments for this block consider the choice of the pixel spacing step between each window to be processed, the minimum number of positive detections (or simply detections) to form a group and the overlap rate between two groups in one or more frames, employed in the spatial and/or temporal analysis.

4.3.1 Spatial Algorithm

Algae detection in an isolated small region (for example, a single 61×61-pixel window) of a frame is most likely a misclassification. In order to suppress such false positives, a spatial analysis of the classifier results was accomplished, taking into account four neighbor windows located at horizontal left, vertical up, diagonal up-left, and diagonal up-right. In order to illustrate the proposed algorithm, consider the image with the classifier results shown in Figure 4.3a, where 16 windows produced positive algae detections (labeled as P). For the evaluation window denoted

in gray in Figure 4.3a, the four neighbor windows employed in the spatial analysis are denoted in red. The algorithm operation is divided into two steps:

- Slide window over the image and assign labels;
- Obtain clusters corresponding to algae detected regions and eliminate false positives.

In the first step, the algorithm assigns labels to the pixels corresponding to algae detection (P). If the pixel of at least one of the four above-defined neighbor windows presented a positive (algae) result, the same label is assigned to it; otherwise, a new label is created. If neighbor windows presented different labels, one of the labels is assigned to the pixel being evaluated. The result of the first step of the proposed algorithm applied to the image in Figure 4.3a is illustrated in Figure 4.3b.

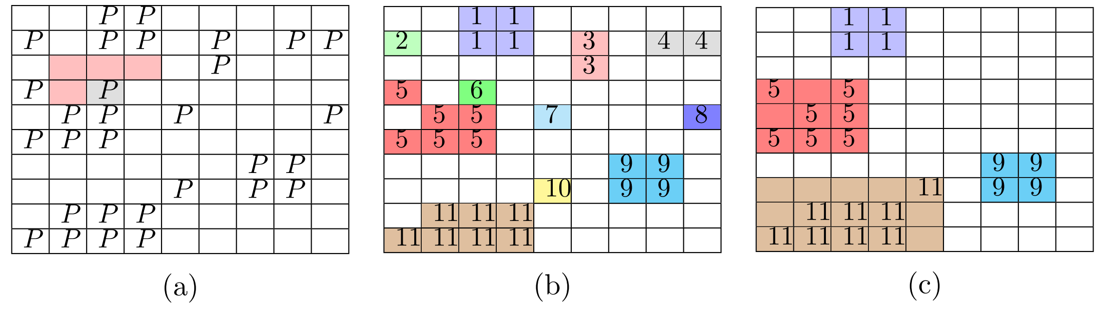


Figure 4.3: Illustration of spatial analysis for false positive removal. In (a), the image obtained from classifier with 30 positive algae detection windows (indicated as P); step space is given by two cells. In (b), First step of the algorithm creates labels for positive algae detection windows based on neighborhood detections. In (c), Second step of the algorithm defines regions with algae and eliminates false positives.

In the second step, different labels of neighbor pixels are replaced by one inside that neighborhood to create a new larger cluster. Next, the minimum and maximum coordinates of each cluster define a rectangular region around the cluster. If the overlap between every two regions is below a chosen threshold value (Ov), the algorithm keeps the clusters separated. Otherwise, the clusters are merged. If a cluster has less than a minimum number N of positive algae detected pixels, then that cluster is eliminated. Additionally, Non-Maximum Suppression (NMS) [76], [77] was employed to remove redundant rectangular regions. Figure 4.3c shows the result after the algorithm was applied. Implementation details are presented in Algorithm 11.

4.3.2 Temporal Algorithm

The spatial analysis algorithm removes satisfactorily from the classifier output image the regions with few sparse false positives. However, due to blurring and other

Algorithm 11 : Spatial analysis for false positive reduction.

Input: I, image from classifier
Input: N, minimum number of positive results
Output: Out, output image with regions of algae detections
First step of the algorithm
1: **for** every positive algae detection pixel $I(x, y)$ **do**
2: label←**FindLabels**($I(x-1, y), I(x-1, y-1), I(x, y-1), I(x+k, y-1)$)
3: **switch** (label)
4: **case** 0**:**
5: $I(x, y) \leftarrow$ newlabel # Assign a new label
6: **case** 1**:**
7: $I(x, y) \leftarrow$ label # Assign the one label found
8: **case** 2, 3, 4**:**
9: $I(x, y) \leftarrow$ label # Assign one of the labels found
10: **AssignEquality**(I) # Assign equality between labels
11: **end switch**
12: **end for**
Second step of the algorithm
13: **JoinLabels**(I) # Replace every assigned equality label
14: **RemoveClusters**(I,N) # Remove small clusters
15: **ComputeRegion**(I) # Compute rectangular detected regions
16: $Out \leftarrow$ **NMS**(I,Ov) # Maintain regions with small overlap

artifacts, a region with false positives may remain being detected in one frame after the spatial analysis, but most likely will not be in the next frames. To reduce this problem, a frame buffer was used to perform a temporal analysis, in order to remove the detected regions without temporal persistence.

Implementation details of the temporal analysis are presented in Algorithm 12, where F is the buffer size, whose appropriate value is evaluated in the Chapter 5. An analysis of the overlap between detected regions (which are within sectors determined by a maximum distance of their centroids for all frames of the buffer) is employed, using the NMS algorithm with percentage threshold parameter Ov.

Figure 4.4 illustrates the application of the proposed algorithm with a frame buffer of size $F = 3$, where the detected regions are delimited in green for the current frame ($n = 0$), in blue for the previous frame ($n = 1$), and in red for the second previous frame ($n = 2$). Figure 4.4a shows the current input image; Figure 4.4b presents the result of the spatial analysis algorithm for the three frames, consisting of 2 detected regions for the current frame, 1 for the previous frame, and 2 for the first stored frame in the buffer.

In order to reduce the false positives of a frame that were not encountered in previous frames, Algorithm 12 employs a procedure based on the centroid distances [32], [31], [30]. After computing the centroids of the detected regions (output of Algorithm 11) of all frames of the buffer (illustrated in Figure 4.4), a matrix that

Algorithm 12 : Temporal analysis for false positive reduction.

Input: F, frame buffer size
Input: Ov, maximum overlap between regions (in percentage)
Input: D, minimum centroid distance
Input: B_n, n-th image of buffer
Input: I, input image from spatial analysis
Output: Out, output image with detected algae regions

```
1:  for n from F-1 to 1 in steps of -1 do
2:     B_n = B_{n-1}                        # Shift images in buffer
3:  end for
4:  B_0 = I                                 # Store new image in buffer
5:  for n from 0 to F-1 in steps of 1 do
6:     C_n = Centroids(B_n)                 # Compute centroid of each cluster of B_n
7:  end for
8:  for n from 0 to F-1 in steps of 1 do
9:     DC = Distances(C_n)                  # Istances between centroids
10: end for
11: if DC(x,y) < D then
12:    R_n = NMS(C_n,Ov)                    # Remain regions
13: end if
14: if R_n appear in all frames of buffer then
15:    Out←DrawRegions(R_n)
16: end if
```

contains the distances between every two centroids (DC matrix) is obtained. Next, the regions with centroid distances smaller than a selected parameter D are merged if the overlap between the corresponding detected regions is below a chosen threshold value Ov, employing the NMS algorithm. Finally, only the regions that appear in all frames of the buffer are kept, as illustrated in Figure 4.4d.

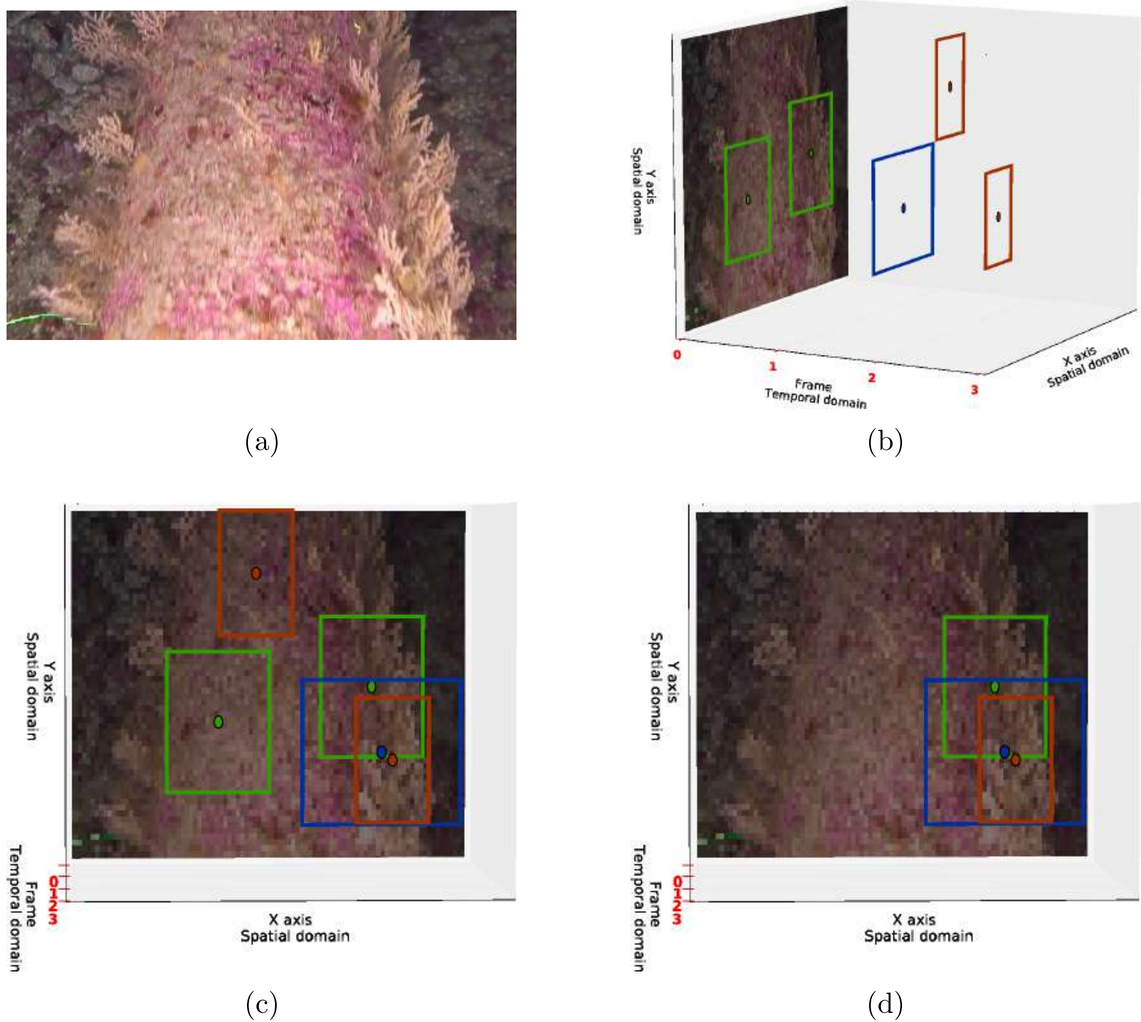


Figure 4.4: Temporal analysis for removing false positives in current frame based on detected regions of previous frames. In (a), Image from current frame is analyzed employing 3 consecutive frames, (b) detected regions after spatial analysis and (c) temporal analysis using 3 time steps in the framebuffer. In (d), Temporal analysis after removing the false positive regions.

Chapter 5

Results and Discussions

This chapter describes the relevant design decisions and implementation details that are taken into account in order to build neural network models for the problem at hand. This chapter also addresses the post-processing algorithm that is proposed. To evaluate the effectiveness of the neural network approach to algae detection, different neural network topologies were tested. They were compared with respect to specific measures, which are described in Sections 5.2 and 5.3. Performance analysis is split into two stages: classifier evaluation and system evaluation.

Section 5.1 describes MLP and CNN design. Section 5.2 describes performance evaluation aiming at optimal classifier configuration (i.e. topology). The classifier analysis stage considers on comparative experiments involving MLPs and CNNs. The remainder of this chapter focuses on the following aspects of this dissertation:

- Neural network topology comparison addressing numbers of neurons in hidden layers;
- Performance differences among different CNN topologies: number of layers, filter sizes, and impact of initialization and regularization;
- False positive rate evaluation for all deep neural network topologies;
- Real image application issues.

5.1 Design Analysis

This section details initial experiments that were carried out to select some hyper-parameter values, and starts the comparison among different neural network topologies. Experiments performed in the following subsections are independent among each others and study several setting for the neural networks in the training phase. This leads us to choose the loss and activation function, regularization techniques,

optimizer, initializer and CNN hyperparameters. Finally, the strategies obtained from results of this section will be used to build and train different topologies described in Section 5.2. We used cross-validation with four folds. The original data set was thus randomly partitioned into four subsets with the same size. In each fold, three of these subsets were used for training, and the remaining subset was used for testing the trained model. Training took 8000 epochs for MLPs and 125 epochs for CNNs. We use the 'CNN+MLP' expression to refer to a neural network having convolutional neural layers connected to a dense layer before the output layer, which is also trained for 125 epochs. Variations on the number of filters was inspired on the VGG net [78], thus making the computational effort at any convolutional layer equal to half the computational effort of the previous layer.

5.1.1 Loss Function

To choose a loss function that is suitable for the problem at hand, in this section we compare MSE and cross-entropy loss functions. For either loss function, the neural network model behavior and the convergence issues have been recently addressed [79], [80]. Experiments have suggested that neural network models based on cross-entropy overcome the models based on MSE, with respect to overall classification error, and that cross-entropy is associated with faster training convergence. Taking into account the multi-dimensional loss function surface, cross-entropy is associated with gradient magnitude (i.e. surface slope) that is larger than MSE gradient magnitude, and so MSE may lead to slower training [79], [80].

5.1.2 Activation Function

Experiments were performed using a simple MLP topology having a single hidden layer, and considering different activation functions for the neurons at that layer. Figure 5.1 shows an MLP with 45 neurons at the hidden layer, and with sigmoidal, hyperbolic tangent, or ReLU activation functions. With respect to training and validation classification (%) error, the best activation function seems to be the hyperbolic. The hyperbolic tangent was selected for the hidden layers in MLPs, while ReLU is selected for all hidden layers in CNNs, including possible dense layers [59], [60]. Recent works often use the ReLU activation function for deep neural network topologies [8], [20], [21], showing classification error that is similar to (or better than) that achieved using the hyperbolic tangent. Vanishing gradient problems associated with the saturated hyperbolic tangent positive values are eliminated if the ReLU activation function is used. Similar works agree with the statements that the hyperbolic tangent is suitable for basic MLP topologies, and that ReLU-type activation functions yield faster convergence in CNN training [60], [81].

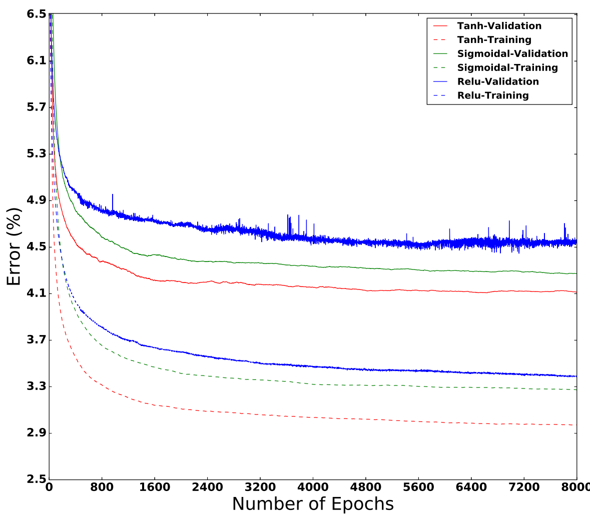


Figure 5.1: Activation function comparison: training (dashed lines) and validation (solid lines) classification error sequences using sigmoidal (green), hyperbolic tangent (red), or ReLU (blue) activation functions.

5.1.3 Regularization

Different regularization methods were applied to different neural network topologies: L2 regularization, dropout, and data augmentation were applied to MLPs, and global average pooling and batch normalization were applied to CNNs. Figure 5.2 shows regularization effects on a CNN+MLP neural network with two convolutional layers and 512 fully-connected neurons. The convolutional layers have 16 and 32 feature maps, respectively. The dense layer uses L2 regularization and dropout (with 0.85 keep probability). The green lines represent a CNN+MLP architecture without regularization in the convolutional layers (dashed line for training classification error and solid line for validation classification error). The blue lines represent the same CNN+MLP architecture with L2 regularization and global average pooling at the convolutional layers. The red lines represent the same CNN+MLP architecture with L2 regularization, global average pooling, and batch normalization at the convolutional layers. Figure 5.2 suggests that global average pooling and batch normalization improves error convergence in neural network training [26], [66]. Batch normalization reduces training time by speeding convergence up. Without regularization, the training loss function may not converge. Global average pooling reduces the number of parameters to be optimized at the connection between the last convolutional layer and the first dense layer. For the experiments presented in Sections 5.2, 5.3, and 5.4, we use L2 regularization, global average pooling, and batch normalization at the convolutional layers. We look forward to obtaining dense layers without too many neurons, to keep the number of parameters relatively small and

to avoid slow training.

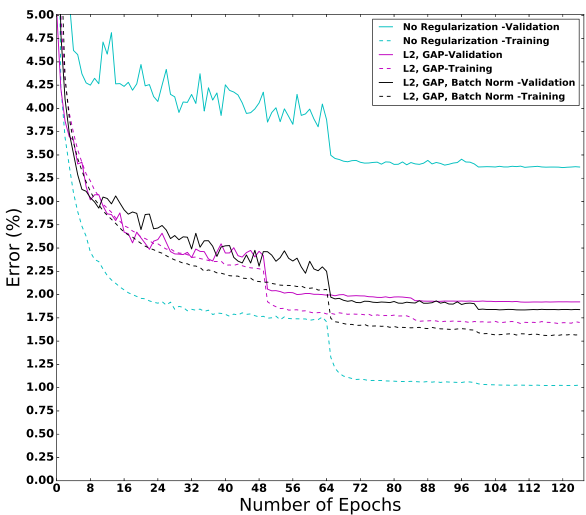


Figure 5.2: Effects of regularization at the convolutional layers of a neural network with CNN+MLP topology: training (dashed lines) and validation (solid lines) classification error sequences using no regularization (cyan); L2 and GAP (magenta); and L2, GAP and batch normalization (black).

5.1.4 Optimizer

For the initial experiments, we use RMSprop and Adam [49], [29]. Figure 5.3 presents training results for a CNN+MLP neural network topology with two convolutional layers, 2× max pooling, and one dense hidden layer with 64 neurons. Each convolutional layer generates 16 feature maps. At the convolutional layers, we use L2 regularization, global average pooling, and batch normalization. At the dense layer, we use dropout with keep probability equal to 0.85. In comparison to RMSprop and Adam for MLP and CNN architectures, momentum yielded much slower convergence and, sometimes, low error results. In Figure 5.3, the RMSprop and Adam training results are similar, with a small advantage for Adam with respect to RMSprop, and the worst results are obtained with momentum. This may be due to the large gradients are not attenuate in the momentum optimizer as the other two methods, generating large updates in the parameters and poor performance.

5.1.5 Initialization

Two initialization techniques were compared, Xavier and Uniform (manually selected). Figure 5.4a shows a same MLP topology (using ***tanh*** as activation function) initialized by means of both techniques, using the best result from uniform initialization. Also, Figure 5.4b shows a simple CNN topology, composed by 2 convolutional

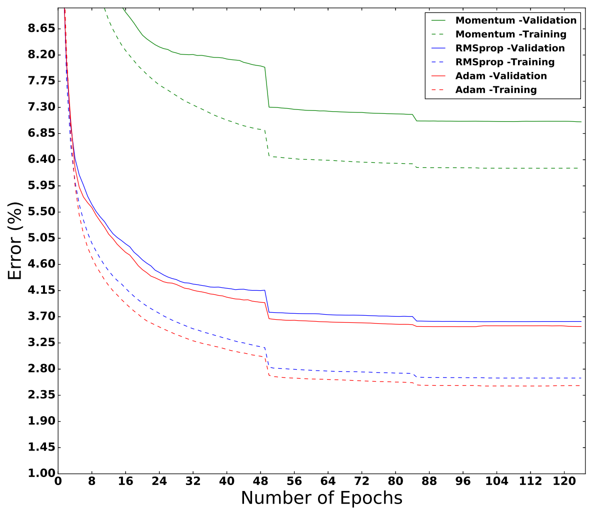


Figure 5.3: Comparison of momentum (green), RMSprop (blue), and Adam (red) training results for a CNN+MLP topology. Training and validation classification error are presented in dashed and solid lines respectively.

layers and 2x2 pooling layers, being initialized through Xavier and Uniform initializations. This analysis indicates that Xavier initialization has an impact in the performance of MLP and CNN topologies. For small MLP architectures the impact is not notorious showed that Xavier initialization increased the performance in deep models, what we confirmed for CNN topologies. Related works state Xavier initialization increased the performance in deep models. Glorot X. et al. (2010) [74] observed this algorithm to initialize performed superiorly than uniform initialization. This also is compared in Mishkin D. et al. (2015) [68] and Hendrycks D. et al (2016) [73] employing other initialization techniques, showing Xavier achieved a suitable performance. Thereby, Xavier initialization was selected for future experiments.

5.1.6 CNN hyperparameters

Kernel size, number of filters (channels) and depth are taken into account. In order to define neural network topologies for training in Section 5.2, the CNN hyperparameters are set using power-of-two number sequences (number of filters and depth) or odd numbers (kernel size). For training, we use L2 regularization, dropout with keep probability equal to 0.9 in every layer, and global average pooling.

- *Kernel size*: to select kernel size, we trained a CNN with two convolutional layers and 2x2 max pooling. Each convolutional layer generates 16 feature maps. Before the output layer, which has two neurons, global average pooling is performed. Kernels with sizes equal to 3×3, 5×5, 7×7, and 11×11 are considered, for both convolutional layers. Figures 5.5a and 5.5b present the

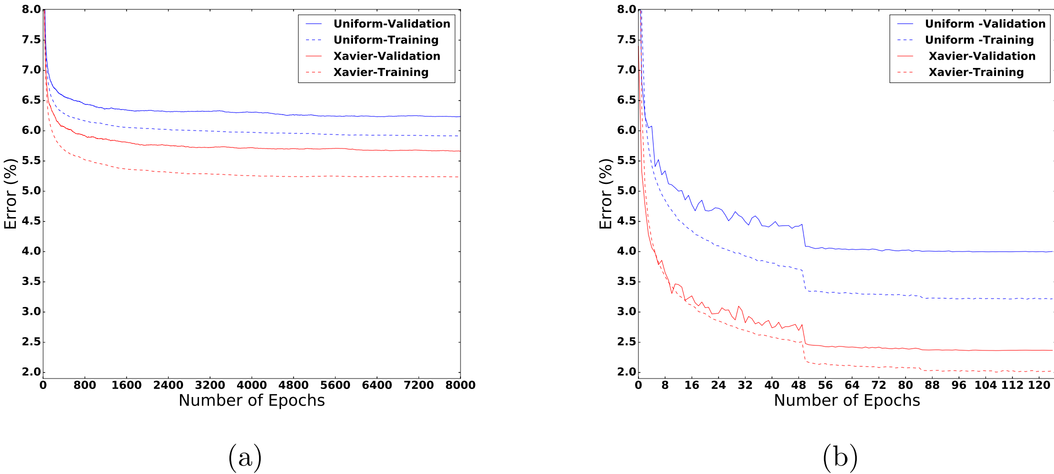


Figure 5.4: Comparison between Xavier and uniform initialization methods. In (a), Xavier and Uniform initialization for a MLP topology. In (b), Xavier and Uniform initialization for a CNN topology. Training and validation classification error are presented in dashed and solid lines respectively.

effects of kernel size variation at the first and second convolutional layers, respectively, on classification error on test data set. The median error values are indicated at the center of the green boxes in the plots, and the average median values (for kernel size variation at the first and second layers) are represented by the blue lines. Similar error results are obtained with all filter sizes, and the computational complexity increases first by a factor $(5/3)^2$ (from kernel size 3×3 to kernel size 5×5), and then by a factor $(7/5)^2$ (from kernel size 5×5 to kernel size 7×7). For simplicity, we thus select kernel size equal to 3×3;

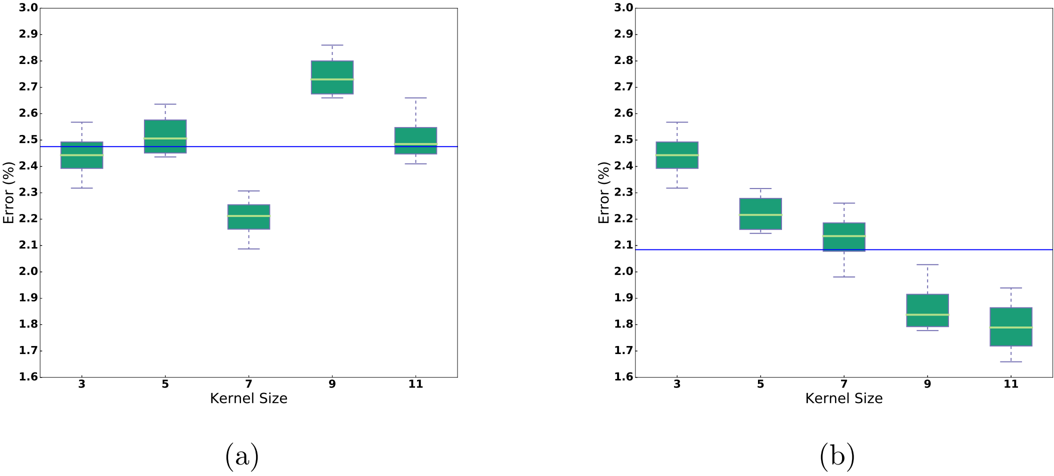


Figure 5.5: Kernel size variation at the first (a) and second (b) convolutional layers of a two-layer CNN. In (a), the average median error is 2.48%. In (b), the average median error is 2.08%.

- *Number of filters*: making variations in the number of filters takes a long training time. We thus varied the number of filters at the first layer, according to powers of two, while keeping the number of filters constant (either 16 or 32) at the second layer of a two-layer CNN. After that, we varied the number of filters at the second layer, according to powers of two, while keeping the number of filters constant (either 16 or 32) at the first layer. Figure 5.6 shows error results on test data set obtained with varying filter sizes. In Figures 5.6a and 5.6b, the number of fixed filters is 16. In Figures 5.6c and 5.6d, the number of fixed filters is 32. The median error values are indicated at the center of the green boxes in the plots, and the average median values are represented by the blue lines. The figures suggest that error is high for CNNs having 16 filters or less at any layer. On the other hand, CNNs having more that 16 filters have an error advantage below 1% with respect to the CNNs having less filters. Increasing the number of filters at the second layer to more than 32 does not improve performance. In the connection from the convolutional layer to the output layer, the large fan-in makes it hard to map 64 neurons (channels) into two neurons. A dense layer may be added between the last convolutional layer and the output layer. To optimize that layer, one might use dropout with a small keep probability;

- Depth: neural network computational complexity increases significantly as its number of layers (depth) increases. We limited CNN complexity to four convolutional layers, each of them having 16 filters. In Figure 5.7a, we compare error values for CNNs with two or three convolutional layers. Using three layers increases the error by around 1% with respect to the error achieved using two layers. For CNNs, when the network depth is increased from three to four, error improves 0.75%. In Figure 5.7b, we compare error values for CNN+MLP topologies having a hidden dense layer with 64 neurons before the output layer. Increasing the depth from two to three improves the error by approximately 0.2%, but the error remains unchanged as the depth is further increased to four. Detailed performance comparisons for different CNN and CNN+MLP topologies are presented in Section 5.2.

5.2 Performance Analysis

In this section, we look for optimal classifier configuration. (i.e. topology and parameters). Test error and runtime are taken into account. Minimum and maximum error values are computed for the training and test data sets, and then compared. The database distribution is detailed in Appendix A. The central processing unit

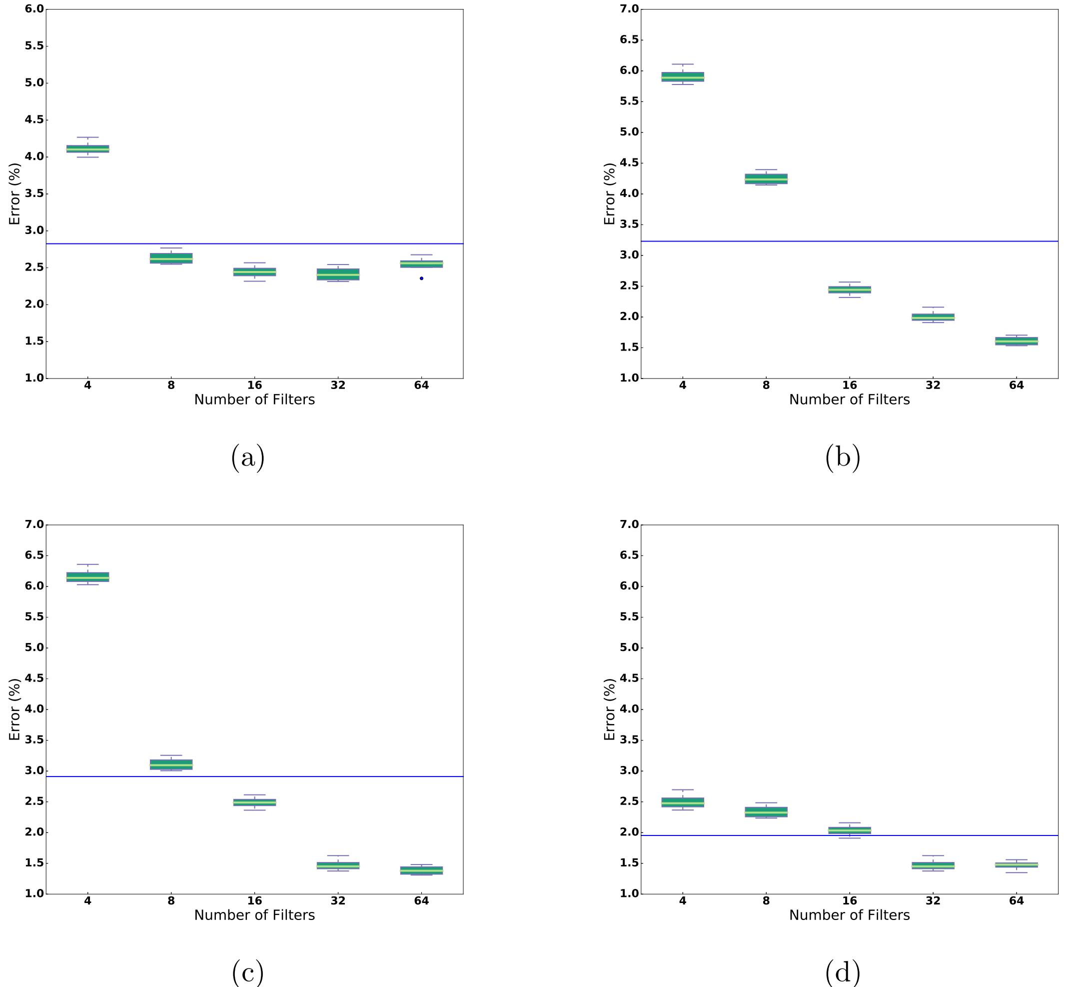
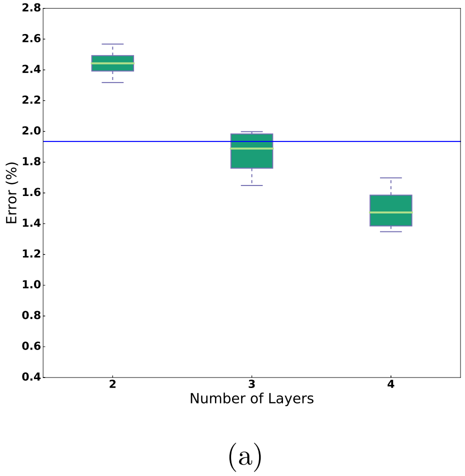


Figure 5.6: Different error results obtained by varying the number of filters at the first or second convolutional layer of a two-layer CNN. In (a) and (c), the number of filters is held constant at the second layer (16 and 32, respectively). The average median error values are 2.83% and 2.91%, respectively. In (b) and (d), the number of filters at the first layer is constant (16 and 32, respectively). The average median error values are 3.23% and 1.95%, respectively.

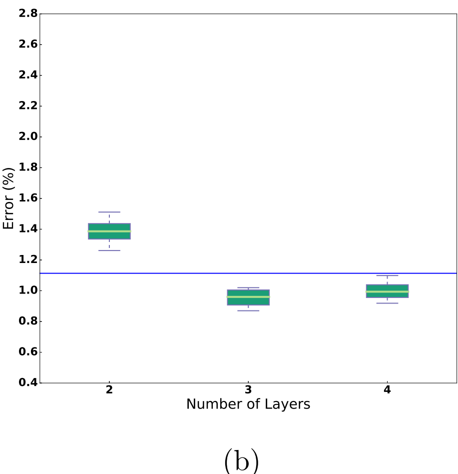
(CPU) used in this dissertation is a 7th-generation i5 processor with 3.5 GHz clock frequency. The graphical processing unit (GPU) used in this dissertation is an NVIDIA GTX 980ti processor with 1.07 GHz clock frequency and 6 GB RAM. For neural network design (training) and test, we use Tensorflow 1.1.0 on the Python application programming interface (API) with GPU support.

5.2.1 MLP Topologies

We initially had hand-crafted feature extractors (see Section 4.1) applied to the database. As a result, six data sets were obtained from the original database. Table 5.1 presents overall success rates (train and test error values) for topologies MLP1,



(a)



(b)

Figure 5.7: Different error results obtained by varying neural network depth, for CNN topologies (a) and for CNN+MLP topologies (b). For CNNs (a), the average median error is 1.93%, and for CNN+MLP topologies the average median error is 1.11%.

MLP2, MLP3, MLP4, MLP5, and MLP6. Those topologies use hyperbolic tangent activation function in a single hidden layer, and sigmoidal activation function at the output layer. The output layer has only one neuron, whose output is thresholded at 0.7 for classification. The first column indicates the number of neurons in the MLP single hidden layer. The loss function is binary cross-entropy. For optimization, we used the Adam algorithm with initial learning rate set to 10^{-3}. The learning rate was progressively reduced according to a pre-established schedule. The training mini-batch size was set to 78, and the overall number of training epochs was set to 8000. Gradient computations are thus performed 624000 times. To avoid overfitting, we used L2 regularization with a 10^{-5} scaling factor for all topologies.

Depending on the data representation corresponding to each feature extractor (for MLPs 1 to 6), the MLP achieves an reduced test error (MLP6, for example), which corresponds to stable convergence given the hyperparameters described in the previous paragraph. Test performance usually improves as the number of neurons is increased from 10 to 200. However, for a large number of neurons at the hidden layer, it is more difficult to make the training process converge. We included dropout in the training, with keep probability set to 0.85, to make convergence more likely. The best classification results (4.33% test error) were obtained by the MLP6 neural network with 200 neurons at the hidden layer. The MLP6 neural network uses Algorithm 10 for feature computation.

For all topologies, the runtime takes from 5 to 8 microseconds per input. The classifier runtime is typically much shorter than the feature extractor runtime. Overall runtime comparisons are provided in Section 5.4.1. Algorithm 10 programming

Topology (Neurons)	MLP1		MLP2		MLP3	
	Train(%)	Test(%)	Train(%)	Test(%)	Train(%)	Test(%)
10	8.86±0.05	10.95±0.23	8.43±0.46	9.25±0.30	14.09±0.08	12.04±0.07
20	7.39±0.06	9.63±0.22	6.72±0.11	7.34±0.13	13.47±0.08	12.05±0.13
30	6.75±0.12	8.82±0.27	6.05±0.08	6.68±0.21	13.59±0.07	12.21±0.04
45	6.29±0.09	8.59±0.08	5.48±0.20	6.20±0.28	12.80±0.09	12.28±0.15
65	5.79±0.05	8.05±0.23	5.06±0.18	5.36±0.21	13.02±0.09	12.42±0.15
75	5.65±0.03	7.90±0.18	4.90±0.13	5.34±0.20	13.29±0.09	12.60±0.20
100	5.34±0.06	7.84±0.45	4.57±0.21	4.97±0.06	13.11±0.05	12.66±0.35
125	5.05±0.05	7.61±0.26	4.28±0.16	4.63±0.13	12.09±0.05	12.57±0.11
200	4.43±0.07	7.06±0.07	3.66±0.05	**4.28±0.14**	2.46±0.05	12.50±0.11
Topologies (Neurons)	MLP4		MLP5		MLP6	
	Train(%)	Test(%)	Train(%)	Test(%)	Train(%)	Test(%)
10	10.70±0.54	16.70±0.27	5.40±0.18	6.98±0.04	5.80±0.06	5.75±0.13
20	12.51±0.43	14.75±0.11	4.38±0.11	6.44±0.10	2.65±0.04	5.21±0.19
30	9.38±0.23	12.37±0.17	3.85±0.03	6.45±0.10	2.02±0.08	5.00±0.19
45	6.12±0.02	11.28±0.29	3.10±0.06	6.31±0.11	1.62±0.04	4.92±0.06
65	7.91±0.09	11.78±0.16	2.62±0.02	6.31±0.10	1.32±0.01	4.83±0.12
75	9.40±0.11	11.52±0.10	3.15±0.07	6.37±0.16	1.13±0.02	4.82±0.06
100	8.10±0.85	10.45±0.24	3.00±0.02	6.18±0.05	1.14±0.06	4.65±0.08
125	6.90±0.03	9.93±0.11	2.40±0.10	6.14±0.17	0.76±0.04	4.45±0.13
200	7.77±0.04	9.87±0.21	2.44±0.09	**6.03±0.08**	0.63±0.04	**4.33±0.03**

Table 5.1: Accuracy comparison among MLP topologies 1 to 6.

was not optimized for runtime.

5.2.2 CNN Topologies

We present the results of CNNs alone (without an MLP at the end), at first, so that the improvement associated with dense layer inclusion becomes clear in Sec. 5.2.3. Several CNN topologies were tested, and they are presented in Table 5.2. The first column indicates the topology design number. The second column shows the number of filters in each convolutional layer. The training and test error values are shown, respectively, in columns three and four. The fifth and sixth columns show GPU and CPU runtimes, and the ratio between the GPU and CPU runtimes is shown in the seventh column. All CNN topologies use 2×2 pooling and global average pooling before the output layer. The output layer is dense and contains only two neurons. All convolutional layers use the ReLU activation function, and the output layer uses softmax activation function. For parameter optimization, we used the cross-entropy loss function. The initial learning rate was set to 10^{-3}, and it was progressively reduced according to a pre-established schedule. The mini-batch size was set to 1574, and the overall number of epochs was set to 125. Gradient computations are thus performed 196750 times. To avoid overfitting, we used L2 regularization with a 10^{-3} scaling factor, global average pooling, and batch normalization.

By comparing Tables 5.1 and 5.2, we notice that the test error values of the smallest CNNs (topologies 1, 2 and 3 in Table 5.2) are just higher than the test error of the best MLP6 in Table 5.1. Some CNNs with three convolutional layers such as

Topology number	Filters number per layer	Train (%)	Test (%)	GPU Time execution (mseg)	CPU Time execution (mseg)	Relation CPU/GPU
1	4, 8	4.25±0.08	6.99±0.33	0.0519	0.3080	5.93
2	4, 32	3.11±0.04	4.90±0.37	0.0500	0.3762	7.53
3	8, 16	2.72±0.03	4.25±0.22	0.0524	0.3866	7.38
4	8, 32	2.06±0.05	3.20±0.12	0.0546	0.4408	8.08
5	16, 4	2.45±0.02	4.03±0.06	0.0550	0.4483	8.15
6	16, 8	1.89±0.06	2.85±0.28	0.0558	0.4645	8.33
7	16, 16	1.39±0.03	2.45±0.09	0.0571	0.4996	8.75
8	16, 32	1.39±0.03	2.49±0.09	0.0598	0.5720	9.57
9	32, 16	1.23±0.03	2.00±0.09	0.0673	0.7289	10.84
10	32, 32	0.95±0.02	1.48±0.65	0.0700	0.8371	11.96
11	32, 64	0.83±0.03	1.47±0.08	0.0777	1.0566	13.61
12	4, 8, 16	2.55±0.03	4.13±0.16	0.0465	0.3279	7.95
13	4, 32, 8	2.09±0.03	2.42±0.05	0.0506	0.4042	7.99
14	4, 32, 16	1.03±0.01	1.97±0.10	0.0507	0.4170	8.23
15	8, 16, 32	1.04±0.03	2.10±0.10	0.0541	0.4319	7.98
16	8, 32, 8	0.87±0.03	2.00±0.05	0.0556	0.4686	8.43
17	8, 32, 16	1.21±0.01	1.99±0.04	0.0561	0.4815	8.58
18	16, 32, 64	0.07±0.16	1.05±0.14	0.0634	0.6947	10.96
19	16, 128, 32	0.55±0.05	**0.80±0.04**	**0.0838**	1.2609	15.05
20	32, 64, 32	0.87±0.07	0.97±0.07	**0.0799**	1.1871	14.85
21	32, 128, 64	0.04±0.15	**0.85±0.12**	0.0992	1.9111	19.26
22	64, 128, 32	0.30±0.01	**0.82±0.02**	0.1268	2.7885	21.99
23	16, 32, 64, 128	0.00±0.00	0.96±0.09	**0.0666**	0.8120	12.20

Table 5.2: Accuracy and runtime comparisons among 23 designed CNNs. Standard deviation values are provided next to mean error values (five folds).

topologies 14, 16 and 18 have error similar to that of CNNs with two convolutional layers such as topologies from 7 to 12. However, in spite of the larger number of layers, those three-layer CNNs run faster and have smaller CPU/GPU runtime ratios than they two-layer counterparts.

CNN topologies 1 to 6 have high test error (see also Figure 5.6). Topologies 7 and 8 are similar with respect to test error and GPU runtime. Topology 9 improves test error by approximately 0.5% with respect to topologies 7 and 8, at the expense of 0.2 ms additional CPU runtime. Topologies 10 and 11, which are the largest two-layer topologies considered, have the best two-layer-based test error: approximately 98.5%. Topologies 15, 16 and 17 achieve test error values 0.5% higher than topologies 10 and 11 do, but topologies 15, 16 and 17 are better than topologies 10 and 11 with respect to GPU and CPU runtime, and with respect to the CPU/GPU runtime ratio. That runtime advantage is due to the longer time taken by the 32 filters at the first layer of CNNs 10 and 11. If the same 3×3 kernel size is used for every layer, then the convolutions at the CNN first layer tend to take longer time than convolutions at subsequent layers. As topologies 12 to 18 in Table 5.2 suggest, the topologies with larger numbers of filters at layers two, three and four tend to have shorter runtime than two-layer topologies with a large number of filters at layer one (topologies 10 and 11, for example). Topology 18 keeps computational complexity approximately the same for every layer. Similarly to what is done in VGG neural

networks [78], at every layer the output volume width or height are half the width or height of the output volume at the previous layer, but the number of filters is twice the number of filters at the previous layer. So, runtime is approximately constant for every layer. Topology 18 improves the test error by approximately 0.9% with respect to the test error values of topologies 15, 16 and 17. Topologies 19 to 22 use larger numbers of filters at layer two. To reduce the number of parameters (synaptic weights) at the connection from the last CNN layer to the output layer (which is dense), topologies 19 to 22 have relatively small numbers of filters at the last convolutional layer. These topologies improve the test error by approximately 0.2% with respect to topology 18, but they require at least 0.02 ms additional GPU time, and at least 0.5 ms additional CPU time. Finally, we notice that a four-layer CNN (topology 23) achieves 0% training error. However, the test error of topology 23 is not larger than the test error values obtained with topologies 19 to 22, and the runtime is slightly larger than topology 18 runtime. Four-layer networks are thus not considered for further (CNN+MLP) training. To have test error references for CNN-based classification, we select the three-layer CNN topologies 19 and 21. To investigate the error improvement that may be obtained by including a dense MLP layer before the output layer (as in Figure 5.7b), other CNN topologies from Table 5.2 are selected too, as described in Section 5.2.3.

5.2.3 CNN+MLP Topologies

Several CNN+MLP topologies were tested, and they are presented in Table 5.3. The first column indicates the topology design number. The second column shows the number of filters in each convolutional layer and the number of neurons in the dense layer. The remaining columns (three to seven) are exactly as described in the first paragraph of Section 5.2.2. Like in the CNN topologies of Section 5.2.2, the convolutional layers always use 2×2 pooling, and global average pooling is always applied immediately after the last convolutional layer. All convolutional and dense layers use the ReLU activation function and the output layer uses softmax activation function. For parameter optimization, we cross-entropy as in Section 5.2.3. Depending on the neural network size, the learning rate was set either to 10^{-3} or to 10^{-4}. The remaining learning rate, mini-batch, and regularization specifications are exactly the same as in the first paragraph of Section 5.2.2, except for the fact that we include dense-layer dropout with keep probability set to 0.5 in the present section.

The CNN+MLP design results are shown in Table 5.3. Some CNN topologies from Table 5.2 were used in these designs. The test error of topologies 7 to 11 from Table 5.2 is improved as an additional dense MLP layer is included in the

Topology number	Filters number per layer	Train (%)	Test (%)	GPU Time execution (mseg)	CPU Time execution (mseg)	Relation CPU/GPU
1	4, 8 + 64	3.45±0.07	4.35±0.05	0.0519	0.3087	5.95
2	8, 16 + 64	1.66±0.05	2.23±0.07	0.0521	0.3879	8.75
3	16, 16 + 64	1.05±0.03	1.34±0.06	0.0572	0.5006	10.97
4	16, 32 + 64	0.24±0.16	1.12±0.05	0.0633	0.6950	10.79
5	32, 16 + 64	0.87±0.01	1.22±0.10	0.0677	0.7304	12.02
6	32, 32 + 64	0.61±0.01	1.10±0.04	0.0698	0.8386	11.95
7	32, 32 + 128	0.62±0.01	1.10±0.03	0.0703	0.8393	13.55
8	32, 64 + 64	0.39±0.02	0.99±0.03	0.0781	1.0584	17.10
9	64, 64 + 128	0.64±0.01	0.92±0.05	0.1057	1.8086	20.56
10	64, 128 + 128	0.39±0.00	0.88±0.02	0.1236	2.5422	7.99
11	8, 16, 32 + 64	0.53±0.02	1.34±0.04	0.0542	0.4329	8.99
12	8, 32, 32 + 64	0.19±0.01	1.09±0.02	0.0567	0.5097	9.94
13	16, 32, 8 + 64	0.19±0.01	1.10±0.03	0.0604	0.6008	11.03
14	16, 32, 64 + 64	0.06±0.02	0.82±0.04	0.0634	0.6999	14.99
15	32, 64, 32 + 64	0.01±0.00	0.92±0.03	0.0799	1.1971	15.60
16	32, 64, 64 + 64	0.11±0.01	**0.65±0.01**	**0.0826**	1.2889	16.92
17	32,64,128+128	0.10±0.03	0.72±0.02	**0.0872**	1.4744	19.40
18	32, 128, 64 + 64	0.06±0.04	**0.61±0.07**	**0.0993**	1.9252	22.23
19	64, 128, 32 + 64	0.09±0.03	**0.80±0.04**	0.1263	2.8082	27.02
20	64,128,256+128	0.06±0.04	**0.59±0.07**	0.1470	3.9714	21.24

Table 5.3: Accuracy and runtime comparisons among 20 designed CNN+MLP topologies. Standard deviation values are provided next to mean error values (five folds).

topologies, as topologies 3 to 8 in Table 5.3 indicate. This improvement comes at insignificant additional runtime cost (the CNN+MLP topologies are approximately 0.01 ms slower on GPU and CPU). Topology 22 from Table 5.2 had achieved one of the best CNN test errors (close to the test error of CNN topologies 19 and 21 in Table 5.2), and so we set the first layer of CNN+MLP topologies 9 and 10 to have 64 filters. Using only two convolutional layers, the hidden dense layer input number becomes large (64 or 128), and 128 neurons are used in the hidden dense layer. With respect to topologies 3 to 8, the test error improvement of CNN+MLP topologies 9 and 10 is insignificant (around 0.1%), so in the next topologies (from 11 on, leaving 17, 19 and 20 out) we avoid more than 32 filters at the first convolutional layer and more than 64 neurons at the hidden dense layer. The three-convolutional-layer CNN+MLP topologies (from 11 on, in Table 5.3) always reach test error below 1.5%. Less than 0.75% test error is achieved by larger topologies (16, 17, 18, and 20). The best test error is achieved by CNN+MLP topology 20. By comparing the best test results in Tables 5.2 and 5.3, we conclude that CNN and CNN+MLP topologies achieve similar test error values, with approximately 0.2% advantage for CNN+MLP 20 with respect to CNN 19.

For the larger topologies (with three or four convolutional layers), the additional parameters (synaptic weights) which are due to the added dense hidden layer usually make training more complicated, even if dropout with 0.5% keep probability is used. For CNN+MLP topologies with two convolutional layers, convergence is faster

than it was for CNN topologies. The test error values are similar for CNN+MLP topologies 16, 17, 18, 19 and 20. For topologies 16, 18, and 19, the number of filters at the last convolutional layer is not as large as in topologies 17 and 20. After global average pooling is performed[1], the synaptic weights of the hidden dense layer correspond to a matrix with size 32×64 (topology 19), or 64×64 (topologies 16 and 18), or even size 128×128 (topology 17), while for topology 20 that size increases to 256×128. Because of the large number of parameters, training is more difficult for topologies 17 and 20.

A comparison between Tables 5.2 and 5.3 also indicates an increase in CNN+MLP runtime with respect to CNN runtime. For some CNN+MLP topologies, the runtime is significantly larger than the runtime of simpler CNN+MLP topologies having similar test error. Then, the topologies with error within 1% of the best topology error may be considered as candidates for CNN-based classifiers with suitable balance between runtime and error. We thus select topologies 6 and 18 as the best designs corresponding to two-layer and three-layer CNN+MLP topologies.

5.3 Other Metrics

This section details other metrics that were carried out to understand the behavior in the dense layer in CNN+MLP, and score metrics used on the best models of each topology.

5.3.1 Relevant inputs

This experiment is conducted in order to discard non-relevant inputs in the networks [82] through the analysis of the most important features from three descriptors (explained in the Chapter 4) for an specific trained model. As shown in Table 5.1, the best accuracy was obtained with topologies that contain 200 neurons in the hidden layer. For such topologies, the relevance of each input generated by MLP1, MLP3 and MLP6 descriptors is presented in Figure 5.8, where the blue lines correspond to the accuracy obtained using all inputs and the dots represent the accuracy obtained replacing each input by its mean value. In Figure 5.8a, the relevance analysis is shown for the MLP1 descriptor, which extracts texture information for the different subbands; for this reason, all inputs are relevant for this feature extractor. In Figure 5.8b, a similar analysis is presented for the MLP3 descriptor, whose LBP features contain local texture information, which often results in redundant areas in the

[1] In [66], the authors conclude that, by reducing the number of parameters to be optimized, global average pooling makes training easier.

output image; thus, some elements of the histogram vector are irrelevant for the trained model. Figure 5.8c shows the relevance of each feature obtained with the MLP6 descriptor, which employs LBP alongside other features. LBP features have little relevant when compared to color descriptor and edge-image entropy. Moreover, shape information are more relevant than texture features, specially for inputs from 8 to 20; this may be due to the fact that several texture features are related to similar areas, what makes some of them irrelevant. This result indicates that shape and color entropy information should generate more features than texture information.

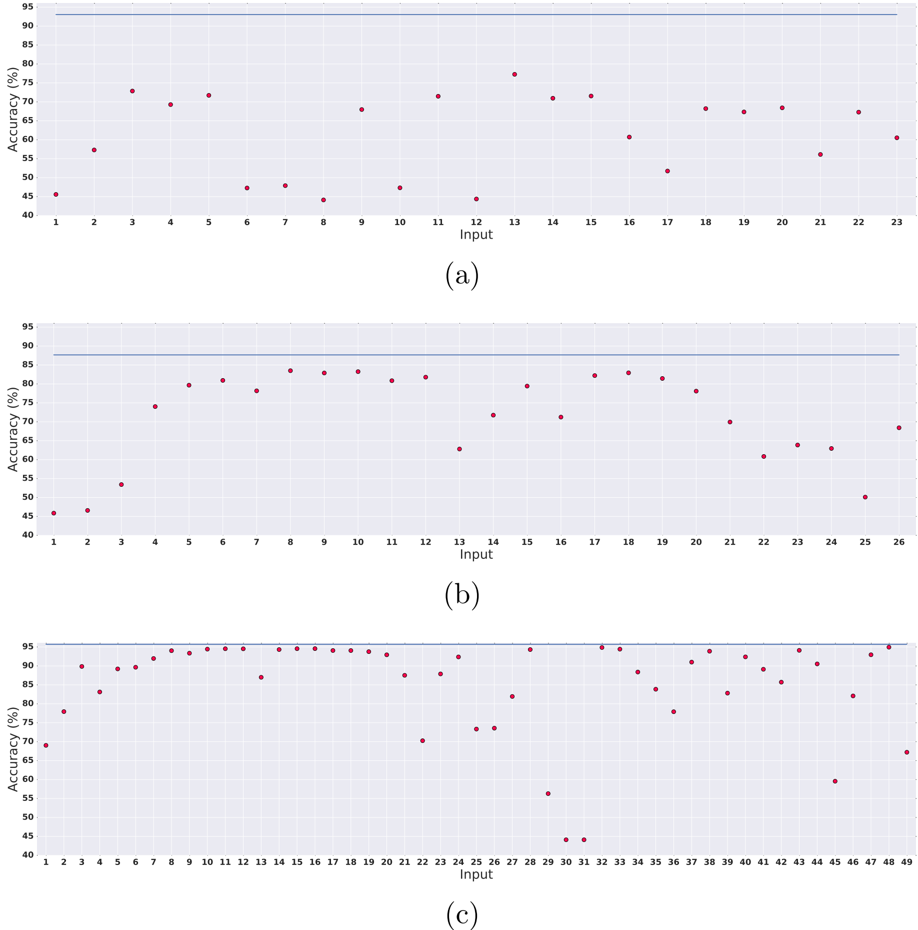


Figure 5.8: Relevant inputs for trained models MLP1, MLP3 and MLP6, obtaining a model accuracy of 93.04%, 87.67% and 95.71% respectively.

From Table 5.3, CNN+MLP 18 is the model that resulted in the best accuracy. In order to perform the relevance analysis of each global average feature map of CNN+MLP 18, the 64 neuron outputs from the GAP operation are fed into the next MLP hidden layer (fully-connected neurons). In Figure 5.9, the relevance analysis results are presented for the 64 features, where the blue line corresponds to the accuracy obtained using all inputs and the dots corresponds to the accuracy obtained by replacing each input by its mean value. It can be observed that, if features 29 and 62 are removed, the accuracy of CNN+MLP 18 increases. Therefore,

it may indicates that the learned features present some feature maps with irrelevant information. In Appendix B, the feature maps of all layer are presented for some input images.

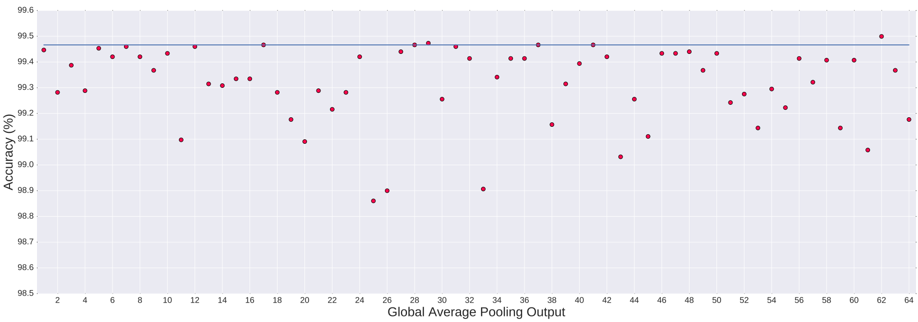


Figure 5.9: Relevant features after global pooling layer for topology CNN+MLP 18, model accuracy 99.47%.

This experiment indicates that when more features are included to a topology, not all have relevant information to perform a task, as observed in Figures 5.8a and 5.8c, which features of MLP1 are relevant while in MLP6 some inputs become irrelevant features. as we observed in the analysis, when the features number increases, some of them may result irrelevant and the feature reduction may improve the error model or optimize the time execution of the network [82]

5.3.2 Score metrics

Since the presence of false positives is more critical than false negatives, the false positive rate (FPR) was used for analysis in specific models selected from MLP, CNN and CNN+MLP topologies. Table 5.4 presents the FPR of each model for the test database.

Architecture	Topology	Accuracy (%)	FPR (%)
MLP1 [83]	200	6.99	6.81
MLP2	200	**4.14**	**4.20**
MLP3	200	12.39	9.01
MLP4	200	9.66	8.78
MLP5	200	5.95	5.69
MLP6	200	**4.29**	**4.67**
CNN 19	16,128,32	**0.76**	**0.36**
CNN 21	32,128,64	0.80	0.59
CNN+MLP 6	32,32+64	1.06	0.83
CNN+MLP 18 [84]	32,128,64+64	**0.53**	**0.30**

Table 5.4: Comparison of false positive rate for selected models.

5.4 Systems Comparison

For comparison purposes, error in videos was measured in two ways: number of false positives, number of positive detections that were lost when temporal and spatial algorithm were applied, and finally, a subjective evaluation of the system results. Moreover, a discussion about the performances of temporal and spatial analysis are presented in this section. Finally, the runtime is analyzed for both CNN-based and MLP-based systems.

5.4.1 False Positives Reduction

Before comparing the performances of the CNN and MLP classifiers, a study was made on the influence of the sliding window step K and the frame buffer size F on the detection error rate and execution time. The results obtained with the CNN classifier applied to 14416 frames (with manually annotated algae regions) are shown in Figures 5.10 and 5.11.

Figure 5.10 shows the results obtained with window shift varying from 1 to 16 pixels, generated in order to determine a step size value that produces a good balance between processing time and performance. Only the spatial analysis (Algorithm 11) was employed in this experiment, with minimum number of pixels in a cluster and overlap threshold between cluster regions equal to $N = 8$ and $Ov = 0.8$, respectively. The region error percentages, measured by the interception-union ratio (IoU) [85] between the delimitation boxes of the detected regions and the ground truth regions, are presented in Figure 5.10a.

The percentage increases in the detected region error, measured by the IoU for K-pixels shift using region for 1-pixel shift as reference, is shown in Figure 5.10b. Figure 5.10c presents the false positive rates (FPR) obtained from 1867 frames that had no algae on the pipelines. Finally, Figure 5.10d shows the runtime of the algorithm for different values of K.

The temporal analysis further improves the FPR due to the fact that the centroids corresponding to the algae on the pipelines present smooth movements. The next experiment was conducted with the purpose of selecting the frame buffer size to be used in Algorithm 2. According to the results of Figure 5.10, $K = 8$ was chosen because the corresponding runtime is small (approximately 0.45 seconds) and the region error is acceptable (around 0.5%). The other parameters of Algorithm 11 were kept equal to those of the previous experiment. The minimum distance between centroids D of Algorithm 11 was set in the range of 10 to 150 pixels, according to the size of the region being analyzed and the number of overlapping regions.

Figure 5.11a shows the FPR (obtained using the same 1867 frames from previous experiment) for different frame buffer sizes F, while Figure 5.11b displays the

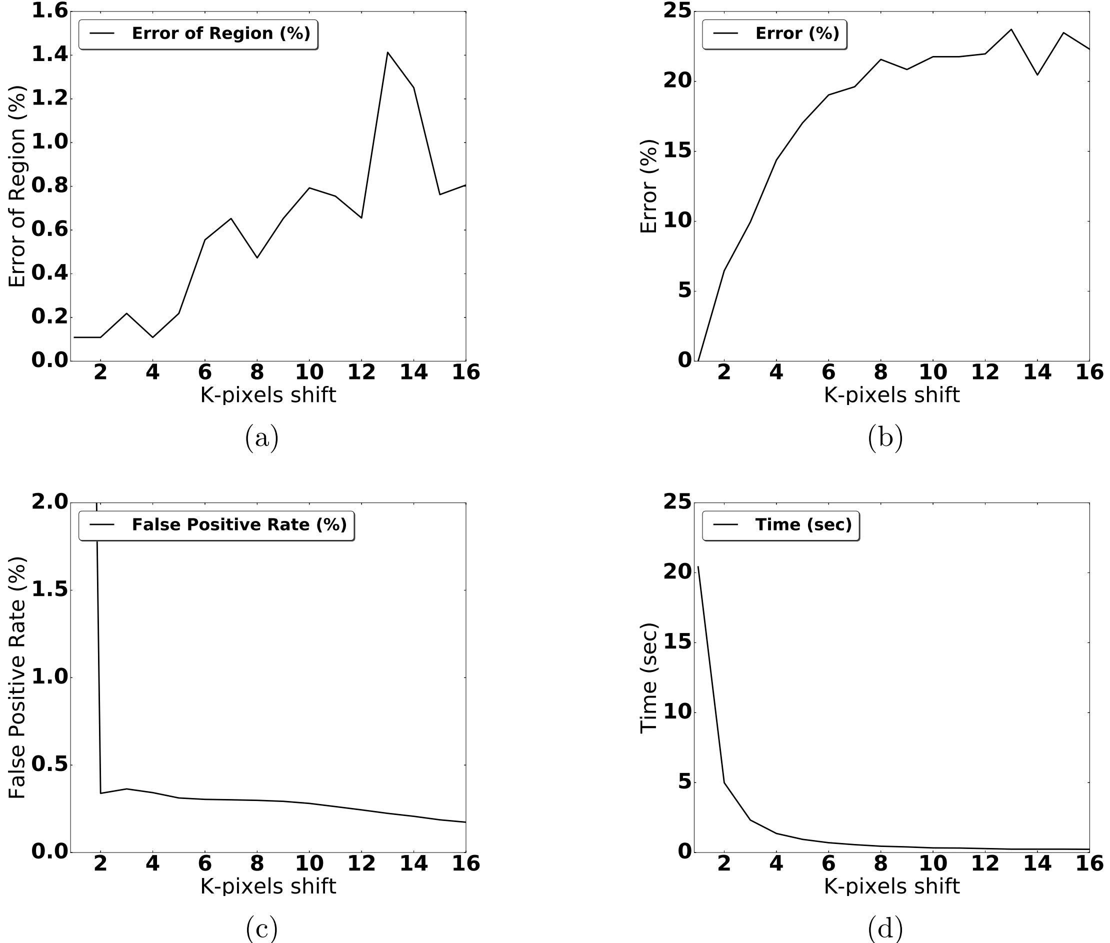
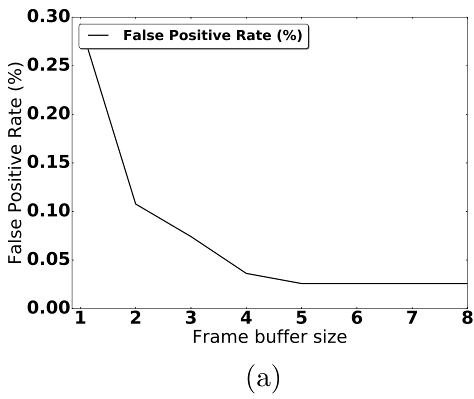


Figure 5.10: Performance and runtime of the algae detection system for different values of the window shift step. In (a), Region error percentage. In (b), Region error increase using 1-pixel shift as reference. In (c), False positive rate. In (d), Algorithm runtime.

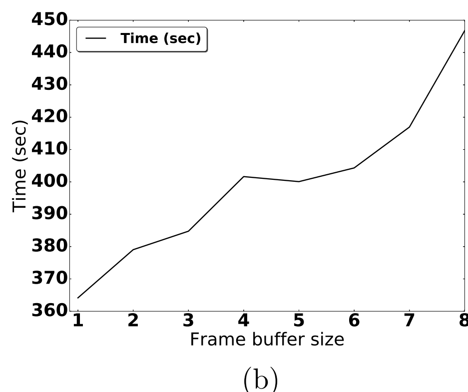
corresponding runtimes. As expected, FPR improves with an increase in buffer size, at the cost of runtime growth. Based on the results of Figure 5.11a, the number of frames F chosen was 2. From Figure 5.11b, it can be observed that as the frame buffer increases so does significantly the system execution time.

5.4.2 MLP-based System

For this system, 200-neurons model from MLP6 showed in Table 5.1 was used to test on real videos, system configuration was established in 8-pixels spatial shift, minimum cluster size 8 and frame buffer size of 4. The framebuffer size is larger for the MLP-based system since false positive regions are detected often even in consecutive 3 frames. The system implemented in CPU only, takes around 10 seconds and 1.5 seconds per frame for descriptor and post-processing algorithms, respectively. Accuracy in videos for this model was 90.05% for 1500 frames in different scenes



(a)



(b)

Figure 5.11: Performance and runtime of the algae detection system for different frame buffer sizes. False positive rate and algorithm runtime for (a) and (b) respectively.

using as reference the best model with shift step K=1 using interception/union as evaluation metric. Figure 5.12 shows outcomes for MLP-based System applied to 6 images.

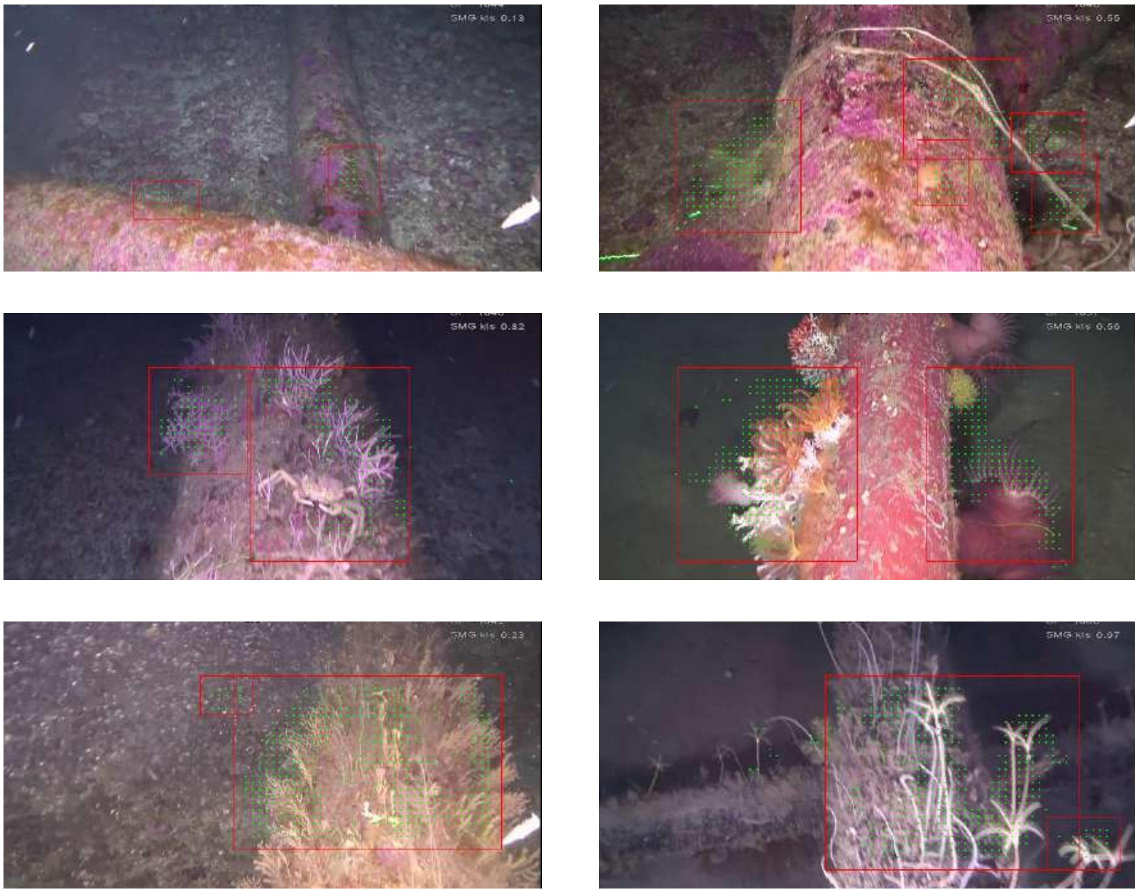


Figure 5.12: Examples of algae detection obtained MLP-based system with 8-pixels spatial shift, minimum cluster size equal to 8 and 2 images in frame buffer.

5.4.3 CNN-based System

The chosen architecture for CNN-based system was CNN+MLP 18 due to its good performance over test set and low FPR. The chosen architecture was applied to the same 6 images previously used to test the MLP-based system. The results are shown in Figure 5.13. To complement the quantitative evaluation, a subjective analysis of the detection results of CNN and MLP methods was performed. It was observed that in several circumstances the MLP algorithm was not able to detect the presence of algae, even when the size of the frame buffer was increased, while the CNN approach detected it with small frame buffer sizes without generating false positives in regions outside the pipeline.

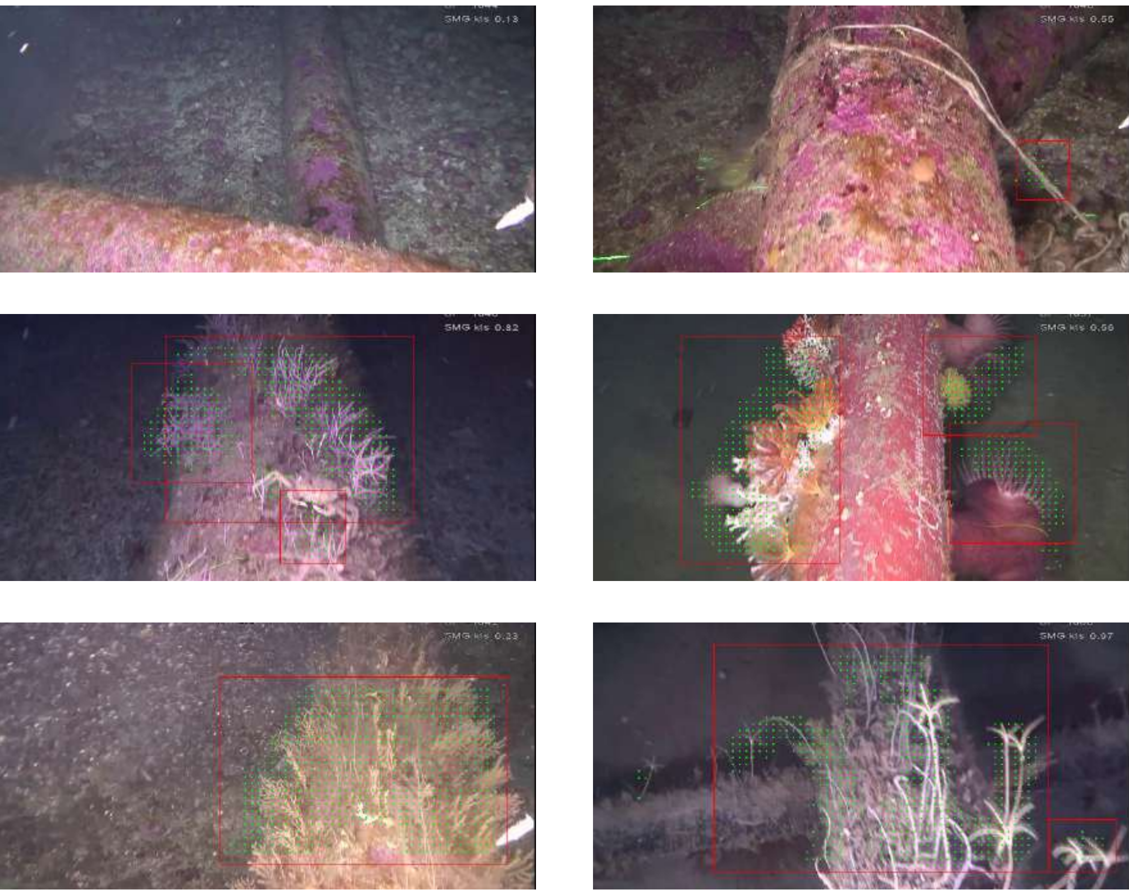


Figure 5.13: Examples of algae detection obtained CNN-based system employing 8-pixels spatial shift, minimum cluster size equal to 8 and 2 frames into frame buffer.

Chapter 6

Conclusions and Future Work

The complexity of automatic underwater pipeline inspection has become large enough so that conventional machine learning algorithms face significant performance limitations in typical applications. In this paper, we compared the performances of conventional MLPs and deep CNNs in an algae detection. The deep CNNs successfully performed high-level feature extraction, and achieved test classification error around 0.6%, whereas the best conventional MLPs achieved 4.3% using different designed features. The classification error improvement comes at the cost of a penalty in processor runtime, but specific libraries such as CUDA and CuDNN allow for improved processing time. The runtime improvement is not available for conventional MLPs, because they have small topologies and they are significantly affected by the data transfer bottleneck even if graphical processors are used.

If regularization techniques are not used, then deep CNN training is impaired by overfitting problems. To obtain generalization corresponding to 0.1% train classification error and 0.6% test classification error, we employed batch normalization, pooling (max pooling at convolutional layers and global average pooling immediately before the fully connected layers), and 10% dropout. Dense layers with 64 neurons allowed fast convergence for all topologies. For some training folds, larger dense layers led either to overfitting or to cost function divergence.

The classification error of the deep CNN test corresponds to acceptable performance in video tests. Occasionally, objects similar to algae led to false-positive rates higher than expected from the design. Often the false-positive results were associated with short time events. In that case, the incorrectly classified windows marked in the video would briefly blink between two correctly classified regions in subsequent frames. To suppress blinking windows and to reduce false-positive results within the same static frame, we proposed temporal and spatial post-processing algorithms. The post-processing algorithms improved the false-positive rate and thus the overall classification error in video tests, especially for blinking windows intervals. Using small frame buffers reduce the processing time, and in this way,

employing a small step size in the spatial post-processing algorithm, the available runtime can be traded off by a reduced false-positive results at the frame level.

Deep neural network structures based on CNNs can be applied to the recognition of several underwater pipeline events other than algae. To further suppress false-positive results at regions outside the pipeline and for objects associated with particular three-dimensional shapes, three-dimensional models typically used in computer vision might be used.

Bibliography

[1] Javier Antich and Alberto Ortiz. Underwater Cable Tracking by Visual Feedback. pages 53–61. 2003.

[2] Marco Jacobi and Divas Karimanzira. Underwater Pipeline and Cable Inspection Using Autonomous Underwater Vehicles. In *2013 MTS/IEEE OCEANS - Bergen*, pages 1–6. IEEE, 2013.

[3] Xu Cao, Xiaomin Zhang, Yang Yu, and Letian Niu. Deep Learning-Based Recognition of Underwater Target. In *2016 IEEE International Conference on Digital Signal Processing (DSP)*, pages 89–93. IEEE, 2016.

[4] Paulo Drews Jr., Vinicius Kuhn, and Sebastiao Gomes. Tracking System for Underwater Inspection Using Computer Vision. In *2012 International Conference on Offshore and Marine Technology: Science and Innovation*, pages 27–30. IEEE, 2012.

[5] Ricardo Pérez-Alcocer, L. Abril Torres-Méndez, Ernesto Olguín-Díaz, and A. Alejandro Maldonado-Ramírez. Vision-Based Autonomous Underwater Vehicle Navigation in Poor Visibility Conditions Using a Model-Free Robust Control. *Journal of Sensors*, 2016:1–16, 2016.

[6] Mariângela Menezes, Carlos E. M. Bicudo, Carlos W. N. Moura, and Et al. Update of the Brazilian Floristic List of Algae and Cyanobacteria. *Rodriguésia*, 66(4):1047–1062, 2015.

[7] Matthew D. Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. pages 818–833, 2013.

[8] Ian Goodfellow Courville, Yoshua Bengio, and Aaron. *Deep Learning*. MIT Press, 2016.

[9] Li Deng and Dong Yu. Deep Learning: Methods and Applications. Technical report, 2014.

[10] Vinh Truong Hoang, Alice Porebski, Nicolas Vandenbroucke, and Denis Hamad. LBP Histogram Selection based on Sparse Representation for Color Texture Classification. In *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, pages 476–483. SCITEPRESS - Science and Technology Publications, 2017.

[11] Bin Liao and Fen Peng. Rotation-Invariant Texture Features Extraction Using Dual-Tree Complex Wavelet Transform. In *2010 International Conference on Information, Networking and Automation (ICINA)*, pages V1–361–V1–364. IEEE, 2010.

[12] Saad Al-Momen. Texture Classification Using Spline, Wavelet Decomposition and Fractal Dimension. *Applied and Computational Mathematics*, 4(1):5, 2015.

[13] Ishrat Jahan Sumana, Guojun Lu, and Dengsheng Zhang. Comparison of Curvelet and Wavelet Texture Features for Content Based Image Retrieval. In *2012 IEEE International Conference on Multimedia and Expo*, pages 290–295. IEEE, 2012.

[14] P. P. Koltsov. Comparative Study of Texture Detection and Classification Algorithms. *Computational Mathematics and Mathematical Physics*, 51(8):1460–1466, 2011.

[15] Jain Stoble B and Sreeraj M. Texture Based Multi-View Face Recognition in Noisy Images Using BRINE Feature. In *2015 International Conference on Computing and Network Communications (CoCoNet)*, pages 806–815. IEEE, 2015.

[16] S. Abirami, V. Ramalingam, and S. Palanivel. Species Classification of Aquatic Plants Using PSVM and ANFIS. *Pattern Recognition and Image Analysis*, 23(2):278–286, 2013.

[17] Roberto Tadeu de Andrade Filho. *Aplicação de Redes Neurais no Controle de Tuberculose Bovina*. PhD thesis, Federal University of Rio de Janeiro, 2016.

[18] Mounir Errami and Mohammed Rziza. Improving Pedestrian Detection Using Support Vector Regression. In *2016 13th International Conference on Computer Graphics, Imaging and Visualization (CGiV)*, pages 156–160. IEEE, 2016.

[19] Erik Marchi, Fabio Vesperini, Stefano Squartini, and Björn Schuller. Deep Recurrent Neural Network-Based Autoencoders for Acoustic Novelty Detection. *Computational Intelligence and Neuroscience*, 2017:1–14, 2017.

[20] John Ray A. Bergado. *A Deep Feature Learning Approach to Urban Scene Classification*. PhD thesis, Enschede, The Netherlands, 2016.

[21] LUIZ GUSTAVO HAFEMANN. *An Analysis of Deep Neural Networks for Texture Classification*. PhD thesis, Universidade Federal do Paraná, 2014.

[22] Hongwei Qin, Xiu Li, Jian Liang, Yigang Peng, and Changshui Zhang. DeepFish: Accurate Underwater Live Fish Recognition with a Deep Architecture. *Neurocomputing*, 187:49–58, 2016.

[23] Sébastien Villon, Marc Chaumont, Gérard Subsol, Sébastien Villéger, Thomas Claverie, and David Mouillot. Coral Reef Fish Detection and Recognition in Underwater Videos by Supervised Machine Learning: Comparison Between Deep Learning and HOG+SVM Methods. volume 10016 of *Lecture Notes in Computer Science*, pages 160–171. Springer International Publishing, Cham, 2016.

[24] Sue Han Lee, Chee Seng Chan, Paul Wilkin, and Paolo Remagnino. Deep-Plant: Plant Identification with convolutional neural networks. 2015.

[25] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[26] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *Arxiv*, pages 1–11, 2015.

[27] Y. Duchi, J., Hazan, E., & Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[28] G. Tieleman, T. and Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude, 2012.

[29] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. 2014.

[30] Zhao Xia Fu and Li Ming Wang. A Simple Algorithm for Video Object Tracking. *Applied Mechanics and Materials*, 303-306:1552–1555, 2013.

[31] Liang Li and Yi Luo. Improved Video Moving Target Tracking Based on Camshift. *American Journal of Computational Mathematics*, 06(04):357–364, 2016.

[32] M. R. Sunitha, H. S. Jayanna, and Ramegowda. Tracking Multiple Moving Object Based on Combined Color and Centroid Feature in Video Sequence. In *2014 IEEE International Conference on Computational Intelligence and Computing Research*, pages 1–5. IEEE, 2014.

[33] Hung-Yu Yang, Pei-Yin Chen, Chien-Chuan Huang, Ya-Zhu Zhuang, and Yeu-Horng Shiau. Low Complexity Underwater Image Enhancement Based on Dark Channel Prior. In *2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*, pages 17–20. IEEE, 2011.

[34] Mark Nixon and Alberto S Aguado. *Feature Extraction and Image Processing for Computer Vision*. Academic Press, 3 edition, 2012.

[35] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, 3 edition, 2007.

[36] Jasjit S Suri by Majid Mirmehdi, Xianghua Xie. *Handbook Of Texture Analysis*. Icp, 1 edition, 2008.

[37] T. Ahonen, A. Hadid, and M. Pietikainen. Face Description with Local Binary Patterns: Application to Face Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):2037–2041, 2006.

[38] Jabal Faizal. Leaf Features Extraction and Recognition Approaches to Classify Plant. *Journal of Computer Science*, 9(10):1295–1304, 2013.

[39] George Azzopardi and Nicolai Petkov. *Computer Analysis of Images and Patterns*, volume 9256 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2015.

[40] Snezana Zekovich and Milan. Hu Moments Based Handwritten Digits Recognition Algorithm. In *Recent Advances in Knowledge Engineering and Systems Science*, 2013.

[41] Dengsheng Zhang and Guojun Lu. Review of Shape Representation and Description Techniques. *Pattern Recognition*, 37(1):1–19, 2004.

[42] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 1 edition, 2012.

[43] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 1 edition, 2007.

[44] Simon O. Haykin. *Neural Networks and Learning Machines*. Pearson, 3 edition, 2008.

[45] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. Chapman and Hall/CRC, 1 edition, 2009.

[46] Lutz Prechelt. Early Stopping - But When? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7700 LECTU:53–67, 2012.

[47] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding Deep Learning Requires Rethinking Generalization. *International Conference on Learning Representations 2017*, pages 1–14, 2016.

[48] Yagang Zhang. *New Advances in Machine Learning*. InTech, 1 edition, 2010.

[49] Sebastian Ruder. An Overview of Gradient Descent Optimization Algorithms, 2016.

[50] Yoshua Bengio. Practical Recommendations for Gradient-Based Training of Deep Architectures. 2012.

[51] Ning Qian. On the Momentum Term in Gradient Descent Learning Algorithms. *Neural Networks*, 12(1):145–151, 1999.

[52] George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. On the Importance of Initialization and Momentum in Deep Learning. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, number 28, pages 8609–8613, Atlanta, USA, 2013. IEEE.

[53] Adam Coates, Blake Carpenter, Carl Case, Sanjeev Satheesh, Bipin Suresh, Tao Wang, David J. Wu, and Andrew Y. Ng. Text Detection and Character Recognition in Scene Images with Unsupervised Feature Learning. In *2011 International Conference on Document Analysis and Recognition*, pages 440–445. IEEE, 2011.

[54] Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. 2012.

[55] Timothy Dozat. Incorporating Nesterov Momentum into Adam. *ICLR Workshop*, (1):2013–2016, 2016.

[56] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington DC, 1 edition, 1962.

[57] Michael Nielsen. Neural Networks and Deep Learning, 2015.

[58] Geoffrey E. Hinton Rumelhart, David E. and R. J. Williams. Learning Internal Representations by Error Propagation. Technical report, University of California, San Diego, 1986.

[59] P Sibi, S Jones, and P Siddarth. Analysis of Different Activation Functions Using back Propagation Neural Networks. *Journal of theoretical and applied Information Technology*, 47(3):1264–1268, 2013.

[60] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical Evaluation of Rectified Activations in Convolution Network. *ICML Deep Learning Workshop*, pages 1–5, 2015.

[61] Raúl Rojas. *Neural Networks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1 edition, 1996.

[62] Y. Le Cun and Y. Bengio. Word-Level Training of a Handwritten Word Recognizer Based on Convolutional Neural Networks. In *Proceedings of the 12th IAPR International Conference on Pattern Recognition (Cat. No.94CH3440-5)*, volume 2, pages 88–92. IEEE Comput. Soc. Press, 1994.

[63] Jayanth Koushik. Understanding Convolutional Neural Networks. *Nips 2016*, (3):1–23, 2016.

[64] C. C. Jay Kuo. Understanding Convolutional Neural Networks with A Mathematical Model. 2016.

[65] Stanford University. Notes from the Stanford CS Class CS231n, 2016.

[66] Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network. *arXiv preprint*, page 10, 2013.

[67] Hui Zou and Trevor Hastie. Regularization and Variable Selection Via the Elastic Net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

[68] Dmytro Mishkin and Jiri Matas. All You Need is a Good Init. *Arxiv*, 2015.

[69] J. Oquab, M., Bottou, L., Laptev, I. and Sivic. Is Object Localization for Free? – Weakly-supervised Learning with Convolutional Neural Networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, III, 2015.

[70] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, pages 1–9, 2012.

[71] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, oct 1986.

[72] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. Efficient BackProp. In *Neural Networks: Tricks of the Trade, this book is an outgrowth of a 1996 NIPS workshop*, pages 9–50. pringer-Verlag London, 1998.

[73] Dan Hendrycks and Kevin Gimpel. Generalizing and Improving Weight Initialization. 2016.

[74] Yoshua Bengio Xavier G. Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[75] Vishakha Metre and Jayshree Ghorpade. An Overview of the Research on Texture Based Plant Leaf Classification. jun 2013.

[76] Rasmus Rothe, Matthieu Guillaumin, and Luc Van Gool. Non-maximum Suppression for Object Detection by Passing Messages Between Windows. pages 290–306. 2015.

[77] Navaneeth Bodla, Bharat Singh, Rama Chellappa, and Larry S. Davis. Soft-NMS – Improving Object Detection With One Line of Code. apr 2017.

[78] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. 2014.

[79] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The Loss Surfaces of Multilayer Networks. 2014.

[80] Pavel Golik Ney, Patrick Doetsch, and Hermann. Cross-Entropy vs. Squared Error Training: a Theoretical and Experimental Comparison. pages 1–5. Citeseerx, 2013.

[81] Bing Xu, Ruitong Huang, and Mu Li. Revise Saturated Activation Functions. pages 1–4, 2016.

[82] Héctor F. Satizábal M. and Andres Pérez-Uribe. Relevance Metrics to Reduce Input Dimensions in Artificial Neural Networks. In *ICANN'07 Proceedings of the 17th international conference on Artificial neural networks*, pages 39–48. 2007.

[83] Edgar Medina, Mariane Rembold Petraglia, and José Gabriel Rodriguez Carneiro Gomes. Neural-Network Based Algorithm for Algae Detection in Automatic Inspection of Underwater Pipelines. In *Advances in Soft Computing: 15th Mexican International Conference on Artificial Intelligence, MICAI 2016, Cancún, Mexico, October 23–28, 2016, Proceedings, Part II*, pages 141–148. Springer International Publishing, 2017.

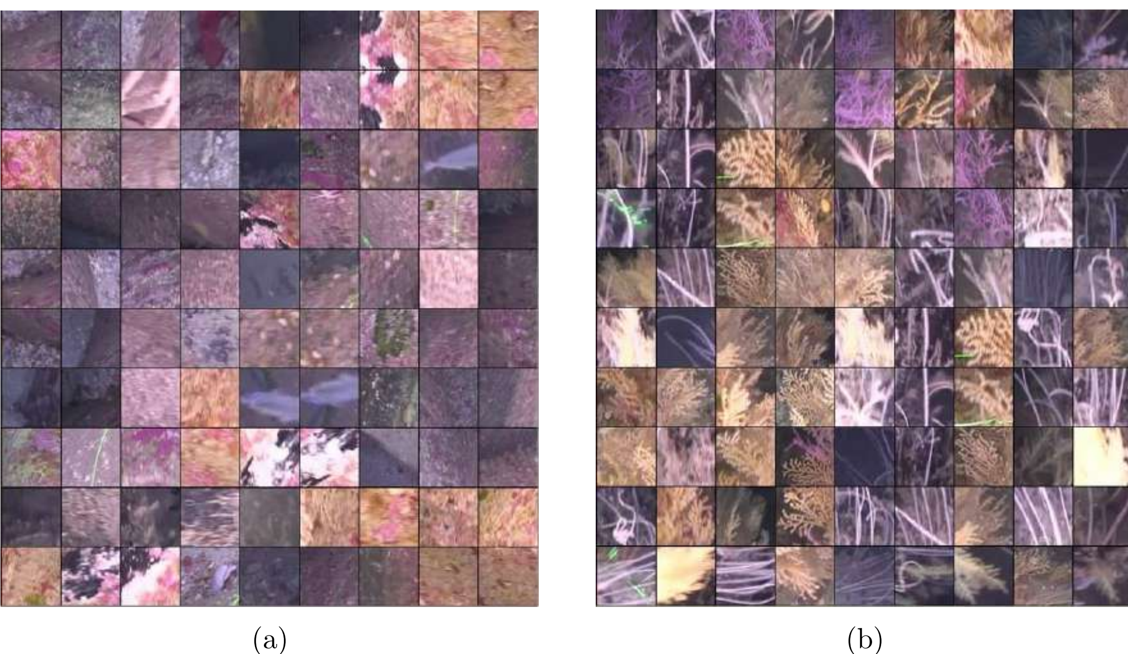
[84] Edgar Medina, Mariane Rembold Petraglia, and José Gabriel Rodriguez Carneiro Gomes. Comparison of CNN and MLP Classifiers for Algae Detection in Underwater Pipelines. *7th International Conference on Image Processing Theory, Tools and Applications (IPTA 2017)*, 2017.

[85] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. 2013.

Appendix A

Algae Database

The database was obtained from videos of underwater pipeline inspection tasks. Each video frame had a size of 1280x720 pixels and the frame rate was 30 fps. To reduce computational cost, image resizing was necessary, resulting in 317x638 pixel images. 41992 samples were annotated manually for two classes: algae and non-algae. The database was split and shuffled at 60% for training, 20% for validation and 20% for testing. Data augmentation was employed after splitting the dataset. Seven extra samples of each original image were included: three images rotated in increments of 90°, the corresponding mirrored images, and the mirror image for 0° rotation. The algae detection algorithms were applied to windows of 61×61 pixels, obtained from the RGB images (video frames) in a sliding mode.



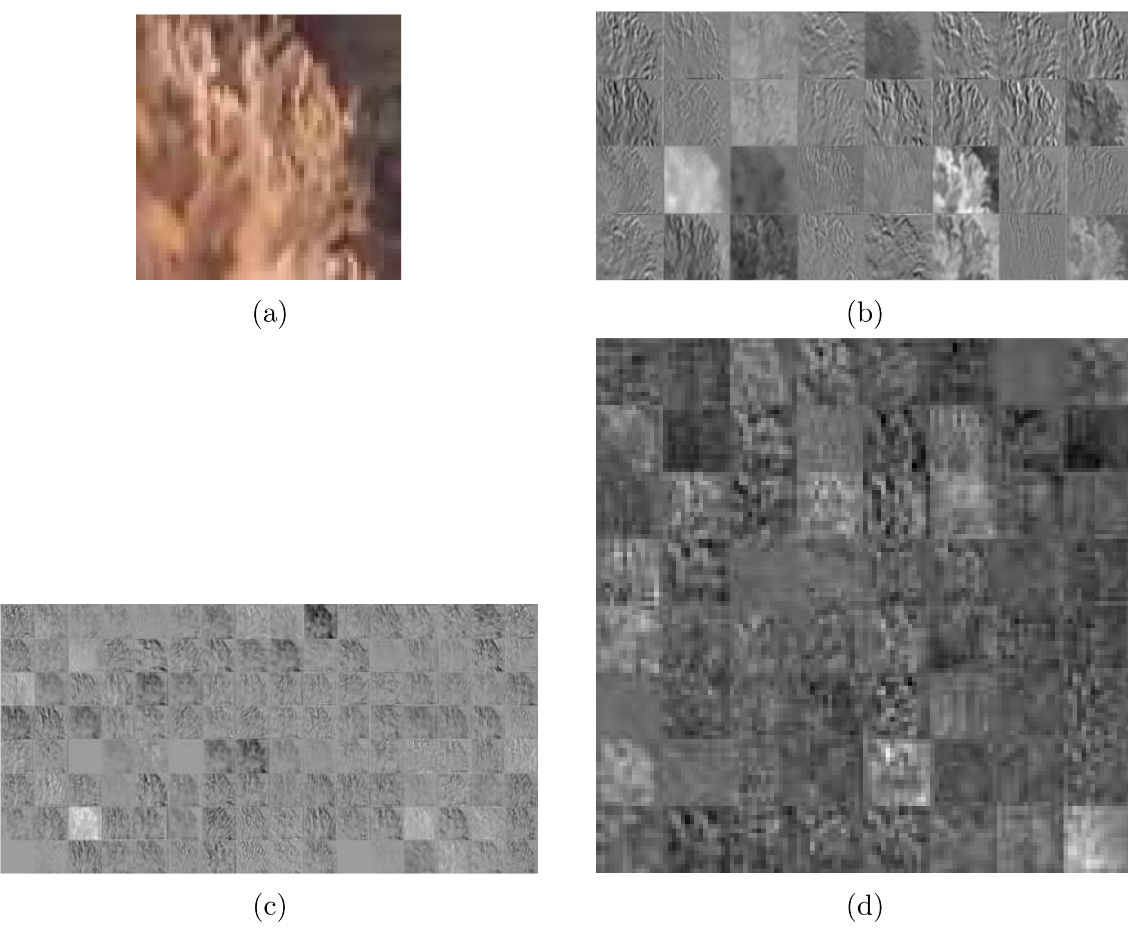
(a) (b)

Figure A.1: Negative (a) and positive (b) samples from image database without data augmentation.

Appendix B

Looking at layers

In order to show the learned features, output from convolutional layer is shown on Figure B.1.



(a) (b) (c) (d)

Figure B.1: Looking at the layers using a positive input sample (a). The first (b), second (c) and third (d) convolutional layers are shown.